FG 2006:
The 11th conference on
Formal Grammar

Malaga, Spain, July 29-30, 2006

**Edited by Paola Monachesi, Gerald Penn,
Giorgio Satta and Shuly Wintner**

**CENTER FOR THE STUDY
OF LANGUAGE
AND INFORMATION**

Preface

Welcome to FG-2006, the 11th conference on Formal Grammar. This year's conference includes 12 contributed papers covering, as usual, a wide range of areas of formal grammar. In addition to the papers included in this volume, the conference features also two invited talks by

- Josef van Genabith, Dublin City University
- Laura Kallmeyer, Universität Tübingen

We are grateful to the members of the Program Committee for their help in reviewing and ranking the twenty four submissions: Anne Abeille (Paris 7, FR), Pierre Boullier (INRIA, FR), Gosse Bouma (Groningen, NL), Chris Brew (Ohio State, US), Wojciech Buszkowski (Poznan, PL), Miriam Butt (Universitaet Konstanz, DE), Alexander Clark (Royal Holloway University, UK), Berthold Crysmann (DFKI, DE), Philippe de Groote (LORIA, FR), Denys Duchier (LORIA, FR), Tim Fernando (Trinity College, IE), Annie Foret (IRISA - IFSIC, FR), Nissim Francez (Technion, IL), Gerhard Jaeger (University of Bielefeld, DE), Aravind Joshi (UPenn, US), Makoto Kanazawa (National Institute of Informatics), Stephan Kepser (Tuebingen, DE), Alexandra Kinyon (University of Pennsylvania, US), Geert-Jan Kruijff (DFKI, DE), Shalom Lappin (King's College, UK), Larry Moss (Indiana, US), Stefan Mueller (Universitaet Bremen, DE), Mark-Jan Nederhof (Max Planck Institute for Psycholinguistics, NL), James Rogers (Earlham College, US), Ed Stabler (UCLA, US), Hans Joerg Tiede (Illinois Wesleyan, US), Jesse Tseng (LORIA, FR), Willemijn Vermaat (Utrecht, NL), Anssi Yli-Jyrae (Helsinki, FI).

We are indebted to all the authors who submitted papers to the meeting, and to all participants of the Confernece.

Paola Monachesi, Gerald Penn, Giorgio Satta and Shuly Wintner, July 2006

Contents

1

Treating clitics with Minimalist Grammars

MAXIME AMBLARD

Abstract

We propose an extension of Stabler's version of clitics treatment for a wider coverage of the french language. For this, we will present the lexical entries needed in the lexicon. Then, we will show the recognition of complex syntactic phenomena as (left and right) dislocation, clitic climbing over modal and extraction from determiner phrase. The main goal of this presentation is the syntax-semantic interface for clitics analyses in which we will stress on clitic climbing over verb and raising verb.

Keywords MINIMALIST GRAMMARS, SYNTAX-SEMANTIC INTERFACE, λ-CALULUS, CLITICS.

Minimalist Grammars (MG) is a formalism which was introduced in Stabler (1997), based on the Minimalist Program, Chomsky (1995). The main idea which is kept from the Minimalist Program is the introduction of constituent move in the formal calculus. Such a "move" operation introduces flexibility in a system which seems to be like Categorial Grammars (CG). We try to recover the correspondance in CG, between syntactic structures and logical forms (interpretative level of the sentence).

This formalisation introduces constraints on the use of move rules, and by this way makes the syntactic calculus decidable. These grammars are lexicalised and all steps of the analysis are triggered by the information extracted from the lexicon: from a sentence, it selects a subset of words. To each word corresponds a sequence of features, and it is the first element of the sequence in the derivation which triggers the next rules.

FG-2006.
Edited by Paola Monachesi, Gerald Penn,
Giorgio Satta and Shuly Wintner.

An advantage of this system is that the structure of the calculus is constant. The coverage of the grammar is extended by adding new elements to the lexicon, never by adding new structural rules. The structural system of these grammars contains only two kinds of rules: move and merge (but extensions exist for both). We refer the reader to Stabler's articles and others for presentation of the use of MG, Stabler (1997), Vermaat (1999).

Clitics are the normal form for pronoun in romance language. The syntactic and semantic behavior of clitics in these languages are complex. For French, clitics often climb over auxiliary verb. Ed Stabler proposes in Stabler (2001) a partial lexicon for french clitics recognition and analysis.

We propose here to extend this lexicon to several well-known linguistic problems. These problems interfere at different levels of analysis. Subject raising is typically a semantic question whereas the clitic climbing over modals is a syntactic question. We propose a new lexicon for its syntactic analysis and then we will show how our semantic interface solves semantic questions.

We will use the description of clitics proposed by Perlmutter in Perlmutter (1971). He proposes a filter to recognize the right order of clitics for romance languages, from where we extract the subfilter:

$$[\{je/tu/\cdots\}|ne|\{me/te/se/\cdots\}|\{le/la/les/\cdots\}|\{lui/leur\}|y|en].$$
[nominative | negative | reflexive | accusative | dative | locative | genitive].

In the first part, we propose an extension of Stabler's version of clitics treatment for a wider coverage of the french language. For this, we will present the lexical entries needed in the lexicon. Then, we will show the recognition of complex syntactic phenomena as (left and right) dislocation, clitic climbing and extraction from determiner phrase. The main goal of this presentation is the last part: the syntax-semantic interface for clitics analyses in which we will stress on clitic climbing over verb and raising verb.

1.1 Lexicon for french clitics

1.1.1 Stabler analysis

Stabler's works on clitics are inspired by Sportiche Sportiche (1992), who proposes the following treatment:

Clitics are not elements moved from position XP^*, but are coreferent to this position. The clitics appearing in the structure bear all the features their co-refering XP^* would bear. Furthermore, clitics do not form an autonomous syntactic object, but they are built into a unit with some host.

In this work, two parts in the cliticization are distinguished. The first one is an empty element which takes an argumental position from the verb. The second is the phonological treatment of the unit - the clitic in the surface structure.

We introduce lexical entries which are phonologically empty but carry special features which need to be unified with features of the phonological part of the clitic. The two different parts are connected by a move operation. If just one of these items occurs in the sentence, derivation fails.

We sum up this treatment in the derivation as follows - the annotation recall the main feature of the word and the annotation on the ϵ recall the word which $\epsilon isthetrace$:

(1) donne ϵ_{-F}
Jean$_{-k}$ la$_{+F}$ donne ϵ_{-F} $\Rightarrow$ Jean$_{-k}$ la donne ϵ_{la}
t$_\epsilon$ Jean$_{-k}$ la donne ϵ_{la}
Jean t$_\epsilon$ t_{Jean} la donne ϵ_{la}.
John t$_\epsilon$ t_{Jean} it gives ϵ_{it}.
John give it.

In more details, the derivation is the following:

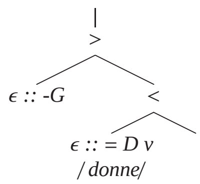
Derivation 1 *Derivation of* Jean la donne.

Lexicon:

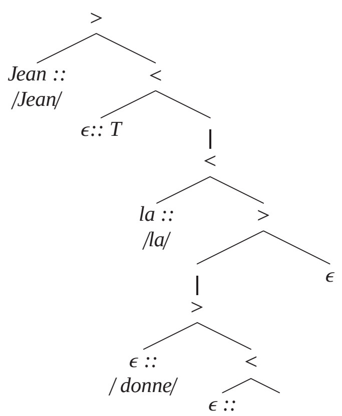
Jean	*D -k*	ϵ	*=T C*	ϵ	*D -k -G*
donne	*V*	ϵ	*=>V =D +k =D v*		
ϵ	*=Acc3 +k T*	*la*	*=v +G Acc3*		

Derivation step by step:

1. *selection of lexical entry : [donne :: V]*
2. *selection of lexical entry : [ϵ :: =>V =D +k =D v] (which adds the syntactic component to the verb).*
3. *head movement. This is a merge between the two previous element where The phonological part of the argument moves to the phonological part of the head.*
4. *selection of lexical entry : [ϵ :: D -k -G]. This is the empty argumental verb position.*
5. *merge.*
6. *There is a licensee "k" in first position, a move operation is triggered. After this step, the derivation tree is :*



7. *selection of lexical entry : [Jean :: D -k].*
8. *merge.*
9. *selection of lexical entry : [la :: =v +G Acc3], the clitic takes part in the derivation.*
10. *merge.*
11. *move : the feature in the empty argument of the verb and the feature in the clitic are cancelled.*
12. *selection of lexical entry : [ϵ :: =Acc3 +K T] - to the end of the derivation.*
13. *merge.*
14. *move : resolution of nominative case :*



15. *selection of lexical entry : [ϵ:: =T C] - empty "complement" position.*
16. *merge ; end of the derivation with feature 'c' : acceptance.*

In his presentation, Stabler proposes a lexicon for accusative, dative and reflexive clitics recognition. He ensures the right order with several verbal types. The analysis is driven by the head and the next clitics to introduce will have to be assigned verbal type as they occur in the Perlmutter filter's order.Stabler uses the SMC - shortest move condition - to exclude the use of a reflexive and an accusative clitics in the same sentence.

1.1.2 Extension: genitive, oblique and nominative clitics

We can extend this first approach of french clitics treatment to other cases, in particular genitive, oblique and nominative. This section will present the lexical entries and the process of acceptance of derivations.

We call "state of a verb" the basic type of the head currently handled. For example, if a verb has a accusative clitic its type will be "Acc".

For genitive and oblique clitics, we just add in the lexicon two new empty argumental positions and a list of possible types for each clitic.

In a first time, we introduce a new verbal type for beginning the cliticization and another where the cliticization is finished. We call them “clitic” and “endclitic”.

Following the Perlmutter filter Perlmutter (1971), the first clitic we have to treat for keeping the right order is the genitive one. We add a genitive state which is connected to the “clitic” state. The verbal state passes to the genitive state by means of a lexical entry the phonological form of which is “en” and carries a licensee feature“en”:

$$[en] :: [clitic <=, +EN, genitif].$$

From this state we pass to all the other states of the cliticization, for example :

$$[le] :: [genitif <=, +G, acc].$$

and if there is only a genitive clitic, we use phonologically empty entry to go to the end of the cliticization.

$$[] :: [genitif <=, finclitic].$$

The “oblique” clitics are treated the same way, except that from “oblique” it is impossible to go back to “genitive”. All lexical entries of this type have a “y” phonological form.

$$[y] :: [clitic <=, +Y, oblique].$$

$$[y] :: [genitive <=, +Y, oblique].$$

In the same way, from oblique we can pass to other possible clitic states, as for example :

$$[le] :: [oblique <=, +G, acc].$$

$$[leur] :: [oblique <=, +F, dat].$$

$$[] :: [oblique <=, finclitic].$$

The nominative case is treated the same way. But the use of this procedure to add new clitic treatment is quadratic in the number of lexical entries. For the nominative pronoun, a discussion could be opened around its clitic state. We consider here that they are clitics.

Another discussion about negative form could rise around the status of the negation marker whose position is after the pronoun.

For the moment, we do not treat the negative form in a right way so we will not include it in this presentation, but we assume that the treatment of nominative clitics is outside the clitic cluster. All the phonological pronoun entries take a verbal form in “endclitic” state and give a new verbal form in “Nom”(inative) state.

We add an empty verb argument which must be included in the derivation before the clitic treatment:

$$[] :: [d, -Subj, -case].$$

The sketch of the analysis is:

- la donne ϵ_{-Nom} ϵ
- Je$_{+Subj}$ la donne ϵ_{-Nom} ϵ
- Je t_{Je} la donne ϵ
  I t_I it give ϵ
  I give it

We add in the lexicon a basic feature "Nom" and the lexical entries of the nominative pronouns, for example:

$$[je] :: [=finclitic, +Subj, Nom].$$

$$[nous] :: [=finclitic, +Subj, Nom].$$

The derivation continues with a phonologically empty entry at the end of the derivation.

$$[] :: [=nom, +case, t].$$

1.2 Recognition of complex phenomena

This treatment of french clitics is simple and can be integrated easily into a larger analysis.

climbing over modal

We treat the clitic climbing over the whole verbal cluster in particular over modal.

The modal is combine with the verb in the inflexion step. The inflexion is treated with head movement and all clitics take their own place after this treatment.

If there are words which must be inserted between the verb and the modal - for sentences with adverbs - we first build the verbal constituent after which we treat the clitics. In this situation, the clitics could climb over the verb constituent or stand after.

For example, in french we can analyse a sentence as:

(2) Je l'ai vu.
I him have seen.
I have see him.

by building the constituent $\underbrace{\text{ai vu}}$. We can extend to sentences with inserted word: "Je l'ai souvent vu" / "I have often see him" with a derivation as :

(3) $\underbrace{\text{ai souvent vu}}$ ϵ_{-Nom} ϵ_{-F}

l'$_{+F}$ $\underbrace{\text{ai souvent vu}}$ ϵ_{-Nom} ϵ_{-F} $\rightarrow$ l'$\underbrace{\text{ai souvent vu}}$ ϵ_{-Nom} ϵ
Je$_{+Nom}$ l'$\underbrace{\text{ai souvent vu}}$ ϵ_{-Nom} ϵ_{-F} $\rightarrow$ Je l'$\underbrace{\text{ai souvent vu}}$ ϵ ϵ
I it often sawn
I often sawn him.

dislocation

Clitic can be a direct recovery of a not-"empty verbal argument", for example in case of nominal dislocation.

There is a non empty verbal argument which must be extracted from the main sentence and become an indirect argument of the verb.

We build a verb with an "argument which must be extracted" - a determiner phrase - DP - must be outside the main sentence. This state is introduced by a pause or comma. It modifies the determiner phrase in two different ways which depend on the side of the extraction:

- it adds a licensee for the left dislocation and cliticization.
- it adds a licensee for cliticization (and nothing for right dislocation)

The main problem is to include in the sentence the right part which will be replaced by the clitic.

Left dislocation: the DP is extracted from the sentence, placed in first position and recovered by a clitic.

(4) Marie le$_i$ voit trop $\underbrace{\text{ce type}_i,}$ $\rightarrow$ Ce type$_i$, Marie le$_i$ voit trop.
That guy, Marie him sees too much.

Lexical entry of modifier of DP.

$$[,] :: [=> d, d, -H, -disloc].$$

The comma will be placed after the DP by a head movement. The first licensee will be cancelled with the licensor of the clitic and the second with another entry that we must add in the classical "comp" entry (This last entry is used to finish the derivation).

$$[] :: [= t, c, +DISLOC].$$

Right dislocation : In this case the determiner phrase is placed at the end of the sentence. For the homogeneity of the mechanism, we add a licencee of recovered by a clitic, and another for the extraction at the end of the sentence.

$$[,] :: [d <=, d, -H, -disloc].$$

The "comp" phase uses a weak move which lets the phonological form of the constituent in its place - here, at the end of the sentence.

(5) Marie le$_i$ voit trop $\underbrace{\text{, ce type}_i}$. $\rightarrow$ Marie le$_i$ voit trop, ce type$_i$.
Marie him sees too much, that guy.

In fact, this extraction seems to be very similar to questions. In questions, an argument of the verb is extracted to take another position in the surface level of the sentence.

Extraction from DP

With the same kind of mechanism, we can extract an argument of any constituent. The determiner phrase can be complex and we extract an argument of the DP. For example:

(6) Pierre en voit la fin - (Pierre voit la fin du film).
Peter of-it sees the end - Peter sees the end of the movie.

We build “la fin ϵ_{-en}” and the cliticization allowed the extraction of the genitive. “Pierre en voit la fin.”

Raising verb

Raising verbs are verbs where one of the arguments is a verb and one of the other arguments is shared by both verbs, like in the sentence:

(7) Il semble le lui donner.
He seems it him give.
He seems give it to him.

where the pronoun “Il” is subject of the two verbs “semble” and “donner”. The second verb must be in infinitive form.

In this case, the sentence has the following structures:

[subject raising_verb clitic infinitive_verb].

A raising verb takes as an argument a verb in infinitive form - with a special inflexion “infinitive” - and without subject. The infinitive inflexion has the lexical entry:

[-inf]::[=>v, verbe].

“verbe” is the feature needed before starting the clitic treatment. A verbal form gets a “verbe” type after the verb receives its inflexion.

The raising verb selects such a “verb”, then a DP subject and then becomes a VP of type “raisingv” which means a VP which has not yet received the inflexion feature and will be able to receive new clitics (in particular pronoun).

For example:

[semble]::[=verbe, =d, raisingv].

This verb should receive its inflexion and its subject. It follows this mechanism until the end of the derivation:

- semble la répare-inf
- semble -ϵ la répare-inf
- Je semble -ϵ la répare-inf

I seem -ϵ it repare-inf

I seem repare it

1.3 Semantic interface

1.3.1 How to use the syntax/semantic interface

From a sentence, we build a formula of higher order logic which represents its propositional structure. We associate to each lexical entry a λ-term and to each syntactic rule an equivalent semantic rule. We assume that the syntactic analysis drives the semantic calculus.

λ-terms application occurs only when an element has no features. We assume the following functions:

$$feat(x) = \begin{cases} 1 \textit{ if the number of feature of } x = 0 \\ 0 \textit{ else} \end{cases}$$

$$sem(x, y) = \begin{cases} 1 \textit{ if feat(x) = 1 or feat(y) = 1} \\ 0 \textit{ else} \end{cases}$$

Syntactic and semantic synchronisation: after any operation in the syntactic calculus, the semantic counter part computes the *sem* function and if $sem(x, y) = 1$, we perform the functionnal application of the two λ-terms. To known which application to perform, we look at the type of the semantic terms.

A semantic tree represents the semantic counter part of the sentence. It is a tree where the leaves are the semantic part of the lexical entries and the inner nodes contain the λ-term built and the direction of the head (of the syntactic part). We use the following notation:

- breaker between direction head and λ-term : $\vdash$.
- application: @

Applications are carried out when syntax allows it, therefore when the function $sem = 1$ for one of the two terms. The following applications are possible:

if sem (λ-term 1, λ-term 2) = 1 — else

$>\vdash$ λ-term 1@ λ-term 2 — $>\vdash$ λ-term 1, λ-term 2

λ-term 1 λ-term 2 — λ-term 1 λ-term 2

If a move operation cancelled the last feature, we represent it by a unary branch in the tree.

Remark. There are two different possibilities for the semantic calculus: either to wait for elements to be completely discharged or to immediately perform the application. But both fail in different cases: immediate application fails in case of “late adjunction” and the other possibility fails in questions treatment. The right solution seems to be intermediate: it consists in determining a subset of features which must be consumed before applications will be performed. For the moment, we choose the first possibility. Later on, we shall do differently but this only involve changes in the *feat* function.

1.3.2 Example of semantic treatment

Clitic semantics

We present a syntactic treatment of clitics in two different parts. One is phonologically empty and is the non empty argument of the verb, the other is syntactically empty but it is a phonological recovery of the first one. The semantical part of the clitic is in the argumental position and this is a free variable which must be bound in the context. The phonological recovery is an identity.

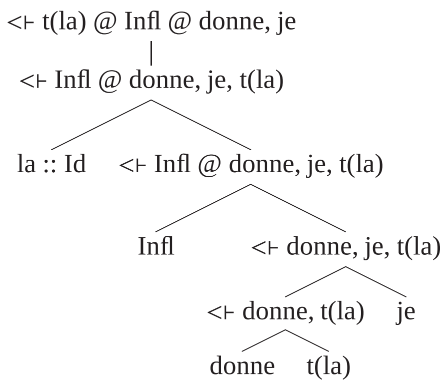
lexical entries	syntactic form	semantic form
la	$dat <= \ +G\ acc$	Id
$t(la)$	$p\ -case\ -G$	x^*

* Free variable, bound in the context - we could use the Bonato algorithm to determinated how this variables are bounded, Bonato (2006)

We briefly present a semantic tree for a clitic treatment:

(8) Jean la répare.
John it repairs.
John repairs it.

In the semantic tree of the part of the clitisization above, we do not represent the identity operator (except for the clitic one).



The last part of the tree is built by a move which creates a link between the phonological part of the clitic and the argumental part.

Over raising verbs

For the semantic calculus, raising verbs are predicates which take a subject and an action as argument. They apply a variable at this action.

We present the analysis of the sentence:

(9) Je semble la réparer.
I seem it repair.
I seem repair it.

The λ-terms, semantic counter-part of lexical entries are:

sembler	$\lambda S\, \lambda v.(seem\ v,\ S\,(v))$
Je	I
ϵ_{la}	Y*
réparer	λ x λ y . repair (y, x)

* this variable is bound in the context

The semantic counter part of the pronoun is a constant refering to the speaker "I". The clitic subject climbs over the raising verb. It can be the subject of both verbs in the sentence due to the semantic structure of the raising verb. If the main verb of the sentence has a subject, the application will not introduce a new variable in the formula, else the main verb needs a variable which stands at the subject place. The raising verb involves this variable by duplication of its subject.

The syntactic analysis builds the following structure:

$$(I@(inflexion@(seem@(la(infinitive@repare)))))$$

which allows the computation of the formula: "la reparer"

$$\lambda x.repair(x, Y)$$

and this term is applied to the raising verb: $\lambda S\, \lambda v.(seem\ v,\ S\,(v))$

$$\lambda v.seem(v, repair(v, Y))$$

At the end of the calculus, we construct the formula:

$$pres(seem(I, repair(I, Y)))$$

where Y is bound in the context.

This is the formula we want to construct for representing the propositionnal semantics of the sentence. The subject clitic syntactically climbs over the main verb, and semantically climbs over the two verbs.

1.4 Conclusion and future work

In this paper, we presented an extension of Ed Stabler's propositions on french clitics in minimalist grammars. The new lexicon makes it possible to treat several other syntactic phenomena, the same way as clitic climbing, e.g. extraction from NP or right and left dislocation.

Then, we proposed a syntax-semantic interface for Minimalist Grammars. The aim of this calculus is to build a formula of higher order logic. The semantic calculus - λ-calculus - is driven by the syntactic one. We emphasize on the way to recognize clitics and semantic implication of climbing with raising verbs.

For future work, we want to integrate the negation into the grammar. We consider that the neg-marker "ne" is a clitic and must be incorporated in the treatment of french clitics. There is another complex phenomenon to consider concerning with clitics in the imperative mode (and negation).

Other cases of raising verbs exist which are more complex, allowing several syntactic clitic climbings as in:

(10) Je la laisse le lui donner.
I her let it (to) him give.
I let her give it to him.

where clitics take place in different orders.

Moreover, we want to continue to modelize the semantic effect of clitics in sentences, in particular for interaction between quantifier scope and clitics, which can introduce ambiguities in sentences like:

(11) Je la laisse tous les lui donner.
I her let all them him give.
I let her gives all to him.

Acknowledgement

The writer like to thank Christian Retoré and Alain Lecomte for crucial supports and one of the anonymous TALN 2006 referees for important comments and examples of this paper.

References

Bonato, R. 2006. *An Integrated Computational Approach to Binding Theory*. Ph.D. thesis, University of Verona.

Chomsky, N. 1995. *The Minimalist Program*. MIT Press, Cambridge.

Perlmutter. 1971. Deep and surface structure constraint in syntax. *Rinehart & Winston* .

Sportiche, D. 1992. Clitic constructions. *Phrase Structure and the Lexicon* .

Stabler, Ed. 1997. Derivational minimalism. *Logical Aspect of Computational Linguistic* .

Stabler, Ed. 2001. Recognizing head movement. *Logical Aspects of Computational Linguistics* Springer-Verlag(2099).

Vermaat, W. 1999. *Controlling movement: Minimalism in a deductive perspective*. Master's thesis, Universiteit Utrecht.

2

Logical Grammars with Emptiness

HOUDA ANOUN & ALAIN LECOMTE

Abstract

The purpose of this paper is to show that we can work in the spirit of Minimalist Grammars by means of an undirected deductive system called $\mathcal{LGE}$, enhanced with constraints on the use of assumptions. Lexical entries can be linked to sequences of controlled hypotheses which represent intermediary sites. These assumptions must be introduced in the derivation and then discharged in tandem by their proper entry which will therefore manage to find its final position: this allows to logically simulate *move* operation. Relevance of this formalism will be stressed by showing its ability to analyze difficult linguistic phenomena in a neat fashion.

Keywords LOGICAL GRAMMARS, MINIMALIST PROGRAM, SYNTAX/SEMANTICS INTERFACE, NON-LINEAR PHENOMENA

2.1 Introduction

Type Logical Grammars (Lambek (1958), Moortgat (1997)) and Minimalist Grammars (Chomsky (1995), Stabler (1997)) are two thriving theories dedicated to natural language analysis. Each one has its intrinsic assets. In fact, the first framework is computationally attractive as it works compositionally and gives the semantics for free. While the second one is based upon a reduced number of rules guaranteeing processing efficiency (Harkema (2000)). Despite their apparent differences, these theories share the same philosophy: they are both lexicalized and present universal sets of rules that allow to explain various linguistic phenomena in multitude of natural languages.

Our goal is to bridge the gap between Categorial and Minimalist Grammars by proposing a new logical formalism $\mathcal{LGE}$ (i.e. Logical Grammars with Emptiness) which captures Minimalist operations (i.e. *merge* and *move*) in a deductive setting. This match between logical framework and Minimalist

FG-2006.
Edited by Paola Monachesi, Gerald Penn,
Giorgio Satta and Shuly Wintner.

Program proves to be fruitful as it gives a better understanding of the different mechanisms involved in Minimalist derivations.
Lecomte, A. and Retoré, C. have already proposed a logical system that simulates Minimalist Grammars: Lecomte and Retore (2001). This latter system is built upon elimination rules for both the slashes and the tensor. The absence of any form of introduction rules leads to an efficient system. However, this restriction is not beneficial insofar as it violates the correspondence between syntactic types and semantic representations. In our new proposal, we want to keep a tranparent interface between syntax and semantics by reintroducing abstraction rules which are applied in a controlled fashion.

Like Abstract Grammars and Lambda-Grammars (de Groote (2001) and Muskens (2003)), $\mathcal{LGE}$ grammars are based upon an undirected logical system which has two interfaces (syntactic-phonetic, syntactic-semantics) owing to *Curry-Howard* correspondence. A syntactic derivation is then a deductive proof of a given sequent built using appropriate inference rules. Both phonetic form and semantic representation result from λ-terms combination which is carried out in parallel with the syntactic derivation, since each deductive rule encapsulates a computational step within the simply typed λ-calculus.

The originality of $\mathcal{LGE}$ stems from the refinement introduced in hypothetical reasoning. Our model aims at preserving the advantages of this technique (e.g. dealing with unbounded dependencies) while constraining its use in order to reduce the size of the search space. Thus, instead of considering freely accessible logical axioms, our system is equipped with finite sequences of consumable controlled hypotheses which are attached to certain lexical entries that are expected to move. Such linked hypotheses represent original sites occupied by their associated entry in the D-structure (i.e. before the displacement operation). They should be introduced during the derivation and then abstracted at the same time by their proper entry which will consequently reach its target. In the case of overt constituent movement, intermediary positions occupied by non-pronounced variables will be systematically replaced by phonetically-empty traces; this explains the choice of our formalism's name: Logical Grammar with Emptiness.
In this paper, we will prove that *move* is a metaphoric notion which can be rigorously formalized using Logic. Moreover, we will show how to capture complex linguistic phenomena (e.g. binding, discontinuity) within $\mathcal{LGE}$ thanks to the combination between Logic power and Minimalist Program ideas.

2.2 Bases of $\mathcal{LGE}$

2.2.1 Types & Terms

In this section, we survey the relevant bases inherent to $\mathcal{LGE}$.
Following earlier proposal by Curry, HB. in Curry (1961) and other more

recent research work: de Groote (2001), Muskens (2003), our system distinguish between two fundamental levels of grammar. The first level is an *abstract* language (tectogrammar) which encapsulates universal *principles*. The second level is a *concrete* one which may contain a range of components (e.g. phenogrammar, semantics) used to encode cross-linguistic variation (e.g. word order, lexical semantics).

Our core logic operates on abstract syntactic types which are inductively defined as follows:

$$\mathcal{T}(\mathcal{A}) := \mathcal{A} \mid \mathcal{T} \multimap \mathcal{T} \mid !\mathcal{T}$$

$\mathcal{A}$ is a finite set of atomic types that contains usual primitives of minimalist grammars (e.g. **c** (sentence), $\mathbf{d}_{acc}$ (noun phrase with accusative case), $\mathbf{d}_{nom}$ (noun phrase with nominative case). Composite types are built using the linear implication $\multimap$ and the exponential operator ! introduced in Girard (1987).

Our framework supports a two-dimensional concrete level dealing respectively with phonetics and semantics. Therefore, we consider two kinds of concrete types, namely Φ-types ($\mathcal{T}_\Phi$) and λ-types ($\mathcal{T}_\lambda$) whose definitions are the following:

$$\mathcal{T}_\Phi := s \mid \mathcal{T}_\Phi \multimap \mathcal{T}_\Phi$$

$$\mathcal{T}_\lambda := e \mid t \mid \mathcal{T}_\lambda \to \mathcal{T}_\lambda$$

The set $\mathcal{T}_\Phi$ is composed of only one atomic type **s** which represents phonetic structures (structured trees), whereas $\mathcal{T}_\lambda$ contains two primitives **e** (individuals) and **t** (truth values). Notice that composite Φ-types are built upon linear implication $\multimap$, whereas composite λ-types use intuitionistic implication $\to$. Both phonetic and semantic representation of expressions are easily defined owing to λ-calculus, thus leading to two sets of terms, namely Φ-terms Λ_Φ and λ-terms Λ_λ. Let Σ be a finite set of phonetic constants and C a finite set of semantic constants. Let $\mathcal{V}_\Phi$ (resp. $\mathcal{V}_\lambda$) be an infinite countable set of typed phonetic (resp. semantic) variables. The set $\Lambda_\Phi(\Sigma)$ of well-typed linear Φ-terms is inductively defined as follows:

1. $\epsilon \in \Lambda_\Phi(\Sigma)$ and ϵ is of type **s**[1]
2. if $\phi \in \Sigma$ then $\phi \in \Lambda_\Phi(\Sigma)$ and ϕ is of type **s**
3. if $(x_\Phi : t_\Phi) \in \mathcal{V}_\Phi$ then $x_\Phi \in \Lambda_\Phi(\Sigma)$
4. if s_1 and s_2 are Φ-terms of type **s** then $s_1 \bullet s_2 \in \Lambda_\Phi(\Sigma)$ and it is of type **s** ($\bullet$ operator is used to combine phonetic structures, it is neither associative nor commutative)
5. if ϕ_1 and ϕ_2 are Φ-terms of types t_1 and $t_1 \multimap t_2$ with no common free variable then $(\phi_1\ \phi_2) \in \Lambda_\Phi(\Sigma)$ and is of type t_2
6. if x_Φ is a variable of type t_1, ϕ_1 a Φ-term of type t_2 and x_Φ occurs free exactly once in ϕ_1 then $(\lambda x.\ \phi_1) \in \Lambda_\Phi(\Sigma)$ and has type $t_1 \multimap t_2$

[1] ϵ represents a phonetically empty element used for traces

$\Lambda_\Phi(\Sigma)$ is provided with the usual relation of β-reduction $\stackrel{\beta}{\Rightarrow}$ enhanced with two additional rewriting rules: $\phi_1 \bullet \epsilon \stackrel{\beta}{\Rightarrow} \phi_1$ and $\epsilon \bullet \phi_1 \stackrel{\beta}{\Rightarrow} \phi_1$.
On the other hand, the set $\Lambda_\lambda(C)$ of λ-terms is defined using a simply typed λ-calculus with two basic operations, namely intuitionistic application and abstraction.

Finally, let $\tau_{\lambda at}$ be a function which assigns a λ-type to each atomic abstract type (we assume for instance that: $\tau_{\lambda at}(\mathbf{c})=\mathbf{t}$, $\tau_{\lambda at}(\mathbf{n})= \mathbf{e}\rightarrow\mathbf{t}$, $\tau_{\lambda at}(\mathbf{d}_{case})= \mathbf{e}$). Two homomorphisms τ_Φ and τ_λ are defined to link abstract types to concrete types as follows:

τ_Φ	τ_λ
$\forall$ t$\in \mathcal{A}$, τ_Φ(t)=**s**	$\forall$ t$\in \mathcal{A}$, τ_λ(t)= $\tau_{\lambda at}$(t)
$\tau_\Phi(t_1 \multimap t_2)= \tau_\Phi(t_1) \multimap \tau_\Phi(t_2)$	$\tau_\lambda(t_1 \multimap t_2)= \tau_\lambda(t_1) \rightarrow \tau_\lambda(t_2)$
$\tau_\Phi(!\ t_1)=\tau_\Phi(t_1)$	$\tau_\lambda(!\ t_1)= \tau_\lambda(t_1)$

2.2.2 Lexical Entries & Controlled Hypotheses

We now introduce the notion of 2-dimensional signs which are the basic units managed by our system. Such signs are of the following form $(l_\Phi,\ l_\lambda) : ty$, where:

- $ty \in \mathcal{T}(\mathcal{A})$ (abstract type)
- $l_\Phi \in \Lambda_\Phi(\Sigma)$ and l_Φ is of concrete type τ_Φ(ty)
- $l_\lambda \in \Lambda_\lambda(C)$ and l_λ is of concrete type τ_λ(ty)

We distinguish between three classes of signs, namely *variable* signs (when $l_\Phi \in \mathcal{V}_\Phi$ and $l_\lambda \in \mathcal{V}_\lambda$), *constant* signs (when $l_\Phi \in \Sigma$ and $l_\lambda \in C$) and *compound* signs (when l_Φ or l_λ is a compound term).

These signs are used to define lexical entries. Lexical entries of $\mathcal{LGE}$ are proper axioms which can be coupled with prespecified sequences of controlled hypotheses. Such hypotheses will occupy intermediary sites, they should be introduced in the appropriate order and then discharged at the same time by their associated entry.
Lexical entries obey the syntax below:

$$\vdash (a_\phi\ ,\ a_\lambda) : ty \quad \multimap\!\!\!\!\!\!\!\!3\quad l_{hyps}$$

where:

- $(a_\phi\ ,\ a_\lambda) : ty$ is a 2-dimensional sign.
- l_{hyps}= $([H_1\ :\ t \vdash H'_1\ :\ t], ..., [H_k\ :\ t \vdash H'_k\ :\ t])$ is a sequence of controlled axioms of length $|l_{hyps}|$=k, ($\forall$ i$\in$\{1..k\}, H_i=$(h_{\phi i}\ ,\ h_{\lambda i})$ and H'_i=$(h'_{\phi i}\ ,\ h_{\lambda i})$ where $h_{\phi i} \in \mathcal{V}_\Phi$ (Φ-variable), $h_{\lambda i} \in \mathcal{V}_\lambda$ (λ-variable) and $h'_{\phi i} \in \Lambda_\Phi(\Sigma)$) .

Lexical entries are classified in two groups: *linked entries* (when k>0) and *free ones* (when k=0). Linked entries are coupled with non-empty sequences of controlled hypotheses. Each hypothesis is encapsulated inside an axiom

'$(h_{\phi i}\,, h_{\lambda i})$:t $\vdash$ $(h'_{\phi i}\,, h_{\lambda i})$:t ' which can be either logical (if $h_{\phi i}= h'_{\phi i}$) or extra-logical (if $h_{\phi i}\neq h'_{\phi i}$). Extra-logical axioms are extremely useful since they represent pronounced variables or phonetically non-empty traces stemming from displacement (e.g. pronouns: he, her ...).
The abstract type **ty** of the lexical entry should verify the following specification:

1. if k=0 then ty is an arbitrary abstract type
2. if k=1 then ty=$t_1 \multimap ... \multimap t_n \multimap$ (t$\multimap$ t')$\multimap$t"
3. otherwise ty=$t_1 \multimap ... \multimap t_n \multimap (!t \multimap t')\multimap$t"

Intuitively, the second (resp. third) point above means that our lexical entry represents a constituent that needs to merge with exactly n (n≥0) expressions of types t_1 ... t_n respectively, and then move once (resp. an unspecified number of times, e.g. cyclic move) to reach its final position.
Finally, a lexicon is nothing else but a finite set of lexical entries $\{e_1, ..., e_n\}$.

Let us illustrate the previous definitions in a concrete example. If we assume that ($whom \in \Sigma$) and ($\wedge \in \mathcal{C}$) then the phonetic behavior and the semantic representation of the relative pronoun '*whom*' can be modelled using the linked entry below:

$$\vdash \begin{pmatrix} \lambda\phi\ \lambda m.\ m \bullet (whom \bullet \phi(\epsilon)) \\ \lambda P\ \lambda Q\ \lambda x.\ P(x) \wedge Q(x) \end{pmatrix} : (d_{acc} \multimap c) \multimap n \multimap n\ \ \multimapinv\ [X : d_{acc} \vdash X : d_{acc}]$$

Our entry is linked to one hypothesis which will occupy the initial position of '*whom*', namely the object of its relative clause (e.g. *(book) whom Noam wrote* _). This assumption will be discharged afterwards by its related entry, thus guaranteeing the combination between the relative pronoun and its subordinate clause. Formal rules that manage this overt displacement will be set forth in the next section.

2.3 Logical simulation of Minimalism

2.3.1 Inference rules

Let $lex=\{e_1, e_2, ..., e_n\}$ be a lexicon. $\mathcal{LGE}$ grammar with lexicon *lex* is based upon a deductive logical system which deals simultaneously with two interfaces (syntactic-phonetic, syntactic-semantic).
Judgments of our calculus are sequents of the following form:

$$\Gamma \vdash (l_\Phi\,, l_\lambda) : ty\ ;\ \mathbf{E}$$

where:

- Γ the context is a finite multiset of 2-dimensional variable signs
- $(l_\Phi\,, l_\lambda)$: *ty* is a 2-dimensional sign

- **E** is a finite multiset containing identifiers of all linked lexical entries that were used in the course of the derivation and whose associated assumptions are not yet discharged

Variable signs included in the context Γ correspond to controlled hypotheses that were introduced in the course of the derivation. Each hypothesis will be marked using a superscript '$\uparrow^i$' which points at the lexical entry to which the assumption is attached (e.g. $x_\Phi^{\uparrow^i}$: hypothesis linked to e_i entry).

The first group of $\mathcal{LGE}$ inference rules are axioms which coincide with derivations' leaves. Figure 1 shows axioms that our system supports[2].

$$\frac{e_i = (\vdash a_\phi : ty \multimapinv l)}{\vdash a_\phi : ty;\ \ if\ l = ()\ then\ \emptyset\ else\ \{e_i\}}\ Lex$$

$$\frac{e_i = (\vdash \_ \multimapinv l_{hyp}) \quad l_{hyp}[j] = (x_\phi : A \vdash y_\phi : A)}{x_\phi^{\uparrow^i} : A\ \vdash\ y_\phi : A;\ \emptyset}\ Ctrl$$

FIGURE 1 Axioms of $\mathcal{LGE}$(lex)

Our core logic includes extra-logical axioms which are extracted from lexical entries owing to rule *Lex*. If the involved entry is linked, then its identifier is added to the multiset **E**. On the other hand, our system excludes the freely accessible identity axiom. Available axioms stem from controlled hypotheses which are coupled with linked lexical entries. These axioms can be introduced in the derivations by means of *Ctrl* rule.

Linked entries in $\mathcal{LGE}$ can be attached to more than one controlled hypothesis. This specification has a very strong linguistic motivation. In fact, it can happen that a constituent occupies more than one intermediary site before reaching its target. Such phenomenon is illustrated for instance in the interrogative sentence '*Which book did John file _ without reading it*?'. In that case, the wh-element '*which book*' occupied two positions before displacement (in the D-structure), namely the complement of the verb *file* and that of the infinitive *without reading*. After movement, the first position becomes empty while the second is occupied by a pronounced variable '*it*'. At the semantic level, both these sites of origin represent the same object.
To account for such non-linear phenomena within $\mathcal{LGE}$, we use the exponential ! whose behavior is described by the usual rules of linear logic (Girard (1987)). Figure 2 presents the derived rules which are relevant to our study.

The generic process that handles the management of controlled hypotheses can be summarized as follows. On the first hand, each assumption of type

[2]For the sake of readability, we focus on the syntactic-phonetic interface

$$\frac{\Delta, x_{\phi}^{\uparrow^i} : B \vdash y_\phi : A;\ \mathbf{E_1}}{\Delta, x_{\phi}^{\uparrow^i} :!B \vdash y_\phi : A;\ \mathbf{E_1}}\ !L \qquad \frac{\Delta, x_{\phi}^{\uparrow^i} :!B, y_{\phi}^{\uparrow^i} :!B \vdash u_\phi : A;\ \mathbf{E_1}}{\Delta, b_{\phi}^{\uparrow^i} :!B \vdash u_\phi[x_\phi := b_\phi, y_\phi := b_\phi] : A;\ \mathbf{E_1}}\ !L^c$$

FIGURE 2 Relevant derived rules for !

ty will get the decorated type *!ty* if it is related to a linked entry e_i which is attached to more than one controlled hypothesis. This transformation is carried out by means of *!L* rule. Intuitively, this means that a hypothesis which represents only one controlled assumption (i.e. of type *ty*) is a particular case of hypotheses that encapsulate *at least* one controlled assumption (i.e. of type *!ty*). On the second hand, contraction rule $!L^c$ is applied to gather all the hypotheses linked to a specific entry e_i in one assumption. This will make it possible to abstract these hypotheses in tandem.
Now, the ground is well prepared to present our logical simulation of Minimalism. It is not difficult to simulate *merge* operation of Minimalist Grammars in a logical setting. In our case, it is nothing else but the direct $\multimap$ elimination ($\multimap$E, cf. Fig.3) which merges two Φ-terms (resp. λ-terms) by means of application operation.

$$\frac{\Gamma \vdash f_\phi : A \multimap B;\ \mathbf{E}_1 \quad \Delta \vdash a_\phi : A;\ \mathbf{E}'_1}{\Gamma, \Delta \vdash (f_\phi\ a_\phi) : B;\ \mathbf{E}_1 \cup \mathbf{E}'_1} \multimap E$$

$$\frac{\Gamma \vdash f_\phi : (C \multimap D) \multimap B; \{e_i\} \cup \mathbf{E}_1 \quad \Delta, c_{\phi}^{\uparrow^i} : C \vdash d_\phi : D; \mathbf{E}'_1}{\Gamma, \Delta \vdash (f_\phi\ (\lambda c_\phi.\ d_\phi)) : B;\ \mathbf{E}_1 \cup \mathbf{E}'_1} \multimap IE\ \ddagger$$

FIGURE 3 Behavior of $\multimap$ connective

Move operation is logically captured thanks to the refined elimination rule $\multimap$IE. This rule allows a constituent to reach its final position by simultaneously discharging its controlled hypotheses which occupied intermediary positions. Our logical formalization of *move* operation shares some ideas with Vermaat's one in Vermaat (1999). In fact, we both consider this operation as the combination of two phases, namely a *merge* step and a *hypothetical reasoning*[3] step (abstraction over sites of origin). Thus, the elements which are expected to move are assigned a higher order type (C$\multimap$ D)$\multimap$ B[4]. Such elements wait to merge with a constituent of type C$\multimap$ D, which results from the abstraction of the intermediary positions in the initial structure (of type D).
However, Vermaat proposal is encoded in a directional calculus: *move* operation is then captured using additional postulates which reintroduce structural

[3]The introduction rule of $\multimap$ is not freely available, it is rather encapsulated inside $\multimap$IE rule
[4]Vermaat considers only the case where D=B

flexibility in a controlled fashion. Our proposal is simpler as it is based upon a flexible undirected calculus. Moreover, it makes it possible to limit the operation of hypothetical reasoning used in displacement which is constrained to a specific amount of hypotheses explicitly given by the lexicon.
Rule $\multimap$IE cannot be applied unless the pre-condition ‡ is verified: all linked axioms coupled with the lexical entry e_i must be introduced in an appropriate order (from the right to the left of l_{hyps} sequence) during the derivation of $(\Delta, c_\phi^{\uparrow^i} : C \vdash d_\phi : D;\ \mathbf{E}'_1)$. Once these assumptions are abstracted, entry e_i regains its final position and is automatically withdrawn from the multiset of unstable lexical entries involved in the derivation.
To formalize the pre-condition ‡, we assume that each assumption $x^{\uparrow^i}$ of the context encapsulates a kind of *history* used to record some relevant data. This additional parameter does not have any impact on our logical system. It only ensures the efficiency of parsing by making the constraint ‡ easier to check. The notation $x^{\uparrow^i}\lfloor\sigma\rfloor$ is used when the history σ of the assumption $x^{\uparrow^i}$ is explicitly given. Otherwise, a function *hist()* can be applied to a given hypothesis $x^{\uparrow^i}$ to get its masked history.
Owing to the contraction rule $!L^c$, each hypothesis $x^{\uparrow^i}$ gathers a sub-set of controlled hypotheses related to entry e_i. The history of an assumption $x^{\uparrow^i}$ can then be encoded as a set of pairs of natural numbers. The first number of each pair represents the index of an involved controlled hypothesis taken from l_{hyps} sequence, while the second one is nothing else but the depth[5] of this hypothesis in the current bottom-up derivation.
Each deduction step updates the history of all assumptions included in the context. For instance, *Ctrl* rule enables the introduction of a specific controlled hypothesis of index j and initiates its history with the single pair (j, 0). On the other hand, rules of Fig.2 and Fig.3 increment[6] the depth of the previously introduced controlled hypotheses. We show below two logical rules enhanced with their explicit management of histories:

$$\frac{e_i = (\vdash \_ \multimapinv l_{hyp}) \quad l_{hyp}[j] = (x_\phi : A \vdash y_\phi : A)}{x_\phi^{\uparrow^i}\lfloor\{(j,0)\}\rfloor : A \ \vdash\ y_\phi : A;\ \emptyset}\ Ctrl$$

$$\frac{\Delta, x_\phi^{\uparrow^i}\lfloor\sigma_1\rfloor :!B, y_\phi^{\uparrow^i}\lfloor\sigma_2\rfloor :!B \vdash u_\phi : A;\ \mathbf{E_1}}{\Delta, b_\phi^{\uparrow^i}\lfloor\sigma_1^{++} \cup \sigma_2^{++}\rfloor :!B \vdash u_\phi[x_\phi := b_\phi, y_\phi := b_\phi] : A;\ \mathbf{E_1}}\ !L^c$$

[5] The number of deduction steps between the introduction of the hypothesis and the current state of the derivation

[6] Incrementing operation is denoted by $()^{++}$: $\{...;(k_i,d_i);...\}^{++}=\{...;(k_i,d_i+1);...\}$

Therefore, the side condition ‡ can be stated formally as follows:

$$‡ \quad iff \quad \begin{cases} \forall k,\ 1 \le k \le |\ l_{hyps}\ | \Rightarrow \ \exists!\ d \mid (k,\ d) \in hist(c_{\phi}^{\uparrow^i}) \\ \forall (k,\ d) \in hist(c_{\phi}^{\uparrow^i})\ \forall (k',\ d') \in hist(c_{\phi}^{\uparrow^i}),\ \ k < k' \ \Rightarrow \ d < d' \end{cases}$$

Finally, it is worth noticing that the constraint ‡ is significant only if the considered derivations are in normal form. Therefore, the absence of both the freely accessible identity axiom and the ⊸I rule is necessary to the success of our approach.

2.3.2 $\mathcal{LGE}$ grammars & generated language

$\mathcal{LGE}$ grammars have two parameters, namely a lexicon and an atomic distinguished type **c**. Let $\mathcal{G}$(lex, c) be a $\mathcal{LGE}$ grammar and ‘**at**’ an atomic syntactic type. We say that a sequence of phonetic constants l=$m_1 m_2 ... m_n$ has abstract type ‘**at**’ within $\mathcal{G}$ (i.e. l∈ $\mathcal{L}_{at}(\mathcal{G})$) iff:

$$\exists\ x_{\phi}, x_{\lambda}\ |x_{\phi} \in struct(m_1, ..., m_n) \ \wedge \ (\vdash (x_{\phi},\ x_{\lambda}) : at; \emptyset)$$

where $struct(m_1, ..., m_n)$ is the range of phonetic structures built using • operator and whose leaves are m_1, m_2, ..., m_n in that order.
Notice that the convergence of derivations requires the introduction and the simultaneous abstraction of all controlled assumptions related to involved lexical entries.
Finally, checking whether a sequence of phonetic constants l is recognized by the grammar $\mathcal{G}$ (i.e. l∈ $\mathcal{L}(\mathcal{G})$) amounts to verifying that l has abstract type **c**.

2.3.3 Example of $\mathcal{LGE}$ derivations

This section is devoted to the study of a hybrid example ‘*More logicians met Godel than physicists knew him*’ which involves two complex linguistic phenomena: binding and discontinuity. The analysis of these phenomena within the directional approach constitutes a real challenge for researchers. All proposed solutions are complex insofar as they led to the extension of the core logic either by defining new syntactic connectives (discontinuity connectives: Morrill (2000)) or by introducing additional packages of postulates as in Hendriks (1995). However, our proposal is able to capture such phenomena in an elegant fashion without using any additional material.
Our treatment of binding follows the same ideas of Kayne. R in Kayne (2002) where he argues that the antecedent-pronoun relation (e.g. between *Godel* and *him*) stems from the fact that both enter the derivation together as a doubling constituent ([Godel, him]) and are subsequently separated after movement. In our system, we account for this idea by defining a linked entry e_1 (cf. Fig. 4) associated with the proper noun *Godel*. This entry requires the introduction of two hypotheses (where the first ‘*him*’ is a pronounced one) which must be discharged at the same time. Therefore, e_1 entry will reach its final position

thus making it possible to semantically link the pronoun with its antecedent.

Id	Φ-terms	λ-terms	Abstract types	Hyps
e_1	$\lambda P_\phi.\ P_\phi$(Godel)	$\lambda P_\lambda.\ P_\lambda$(**Godel**)	$(!d_{acc} \multimap c) \multimap c$	[X:d_{acc}⊢X:d_{acc}], [X':d_{acc}⊢him:d_{acc}]
e_2	logicians	**Logician**	n	()
e_3	physicists	**Physicist**	n	()
e_4	λx. λy. (y•(met•x))	λx. λy. **Meet**$_{Past}$(y,x)	$d_{acc} \multimap d_{nom} \multimap$ c	()
e_5	λx. λy. (y•(knew•x))	λx. λy. **Know**$_{Past}$(y,x)	$d_{acc} \multimap d_{nom} \multimap$c	()
e_6	λx. λy. λ P. λ Q. ((more•y)•Q(ϵ))• (than•(x•P(ϵ)))	$\lambda P_1.\ \lambda Q_1.\ \lambda P_2.\ \lambda Q_2.$ **More**(λx. Q_1(x)∧Q_2(x), λx. P_1(x)∧P_2(x))	n⊸n⊸ $(d_{nom} \multimap c) \multimap$ $(d_{nom} \multimap c) \multimap$c	()

FIGURE 4 Example of $\mathcal{LGE}$ lexicon

On the other hand, we capture discontinuity by gathering the different components of a discontinuous expression in the same lexical entry. For instance, entry e_6 defines the phonetic and semantic behavior of the discontinuous constituent (*more ... than*).

We present, in the following, the main steps of our example's analysis. For the sake of legibility, the bottom-up derivation tree is split into different key parts which will be commented on progressively.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \begin{pmatrix} \lambda x.\, \lambda y.\, y \bullet (knew \bullet x) \\ \lambda x.\, \lambda y.\, \mathbf{Know}_{Past}(y,x) \end{pmatrix} : d_{acc} \multimap d_{nom} \multimap c; \emptyset}\, Lex \qquad \dfrac{}{\begin{pmatrix} x_{\Phi}^{\uparrow 1} \\ x_{\lambda}^{\uparrow 1} \end{pmatrix} : d_{acc} \vdash \begin{pmatrix} him \\ x_\lambda \end{pmatrix} : d_{acc}; \emptyset}\, Ctrl}{\begin{pmatrix} x_{\Phi}^{\uparrow 1} \\ x_{\lambda}^{\uparrow 1} \end{pmatrix} : d_{acc} \vdash \begin{pmatrix} \lambda y.\, y \bullet (knew \bullet him) \\ \lambda y.\, \mathbf{Know}_{Past}(y, x_\lambda) \end{pmatrix} : d_{nom} \multimap c; \emptyset}\, \multimap E}{\begin{pmatrix} x_{\Phi}^{\uparrow 1} \\ x_{\lambda}^{\uparrow 1} \end{pmatrix} : d_{acc} \vdash \begin{pmatrix} \lambda Q.\, ((more \bullet logicians) \bullet Q(\epsilon)) \bullet (than \bullet (physicists \bullet (\epsilon \bullet (knew \bullet him)))) \\ \lambda Q_2.\, \mathbf{More}(\lambda x.\, \mathbf{Logician}(x) \wedge Q_2(x)\ ,\ \lambda x.\, \mathbf{Physicist}(x) \wedge \mathbf{Know}_{Past}(x, x_\lambda)) \end{pmatrix} : \begin{matrix} (d_{nom} \multimap c) \\ \multimap c; \emptyset \end{matrix}}}{\begin{pmatrix} x_{\Phi}^{\uparrow 1} \\ x_{\lambda}^{\uparrow 1} \end{pmatrix} :!d_{acc} \vdash \begin{pmatrix} \lambda Q.\, ((more \bullet logicians) \bullet Q(\epsilon)) \bullet (than \bullet (physicists \bullet (\epsilon \bullet (knew \bullet him)))) \\ \lambda Q_2.\, \mathbf{More}(\lambda x.\, \mathbf{Logician}(x) \wedge Q_2(x)\ ,\ \lambda x.\, \mathbf{Physicist}(x) \wedge \mathbf{Know}_{Past}(x, x_\lambda)) \end{pmatrix} : \begin{matrix} (d_{nom} \multimap c) \\ \multimap c; \emptyset \end{matrix}}$$

The derivation above starts by introducing the last controlled hypothesis (i.e. the assumption representing the accusative pronoun *him*) of the sequence attached to e_1 entry. This hypothesis, then, merges with lexical entry e_5 by means of ⊸E rule. On the other hand, a partial derivation is built by consecutively combining entry e_6 with entries e_3 and e_2. The resulting sequent then merges with the previous one. The last deduction step does nothing but decorating the type of the introduced hypothesis by a ! marker in order to express its ability to gather with the other controlled hypothesis linked to its proper entry. At this stage of analysis, only the second controlled hypothesis of e_1 has been used. Moreover, it was involved in exactly three deduction steps after its introduction, so we can deduce that its current history is: hist($x^{\uparrow^1}$)={(2,3)}.

$$
\dfrac{
\dfrac{}{\vdash \begin{pmatrix} \lambda x.\, \lambda y.\, y \bullet (met \bullet x) \\ \lambda x.\, \lambda y.\, \mathbf{Meet}_{Past}(y,x) \end{pmatrix} : d_{acc} \multimap d_{nom} \multimap c; \emptyset}\, Lex
\qquad
\dfrac{}{\begin{pmatrix} z_{\Phi}^{\uparrow^1} \\ z_{\lambda}^{\uparrow^1} \end{pmatrix} : d_{acc} \vdash \begin{pmatrix} z_{\Phi} \\ z_{\lambda} \end{pmatrix} : d_{acc}; \emptyset}\, Ctrl
}{
\dfrac{\begin{pmatrix} z_{\Phi}^{\uparrow^1} \\ z_{\lambda}^{\uparrow^1} \end{pmatrix} : d_{acc} \vdash \begin{pmatrix} \lambda y.\, y \bullet (met \bullet z_{\Phi}) \\ \lambda y.\, \mathbf{Meet}_{Past}(y, z_{\lambda}) \end{pmatrix} : d_{nom} \multimap c; \emptyset}{\begin{pmatrix} z_{\Phi}^{\uparrow^1} \\ z_{\lambda}^{\uparrow^1} \end{pmatrix} :!d_{acc} \vdash \begin{pmatrix} \lambda y.\, y \bullet (met \bullet z_{\Phi}) \\ \lambda y.\, \mathbf{Meet}_{Past}(y, z_{\lambda}) \end{pmatrix} : d_{nom} \multimap c; \emptyset}\, !L
}\, \multimap E
$$

In this second part of analysis, the first controlled assumption linked to e_1 entry is introduced. Then, it merges with e_4 entry which represents the past form of the transitive verb *meet*. This branch of the derivation ends by a !L step like the previous one. We can easily check that, at this point of the derivation, the history of $z^{\uparrow^1}$ assumption is nothing else but hist($z^{\uparrow^1}$)={(1,2)}.

$$
\dfrac{
\dfrac{\dots}{\begin{pmatrix} x_{\Phi}^{\uparrow^1} \\ x_{\lambda}^{\uparrow^1} \end{pmatrix} :!d_{acc}, \begin{pmatrix} z_{\Phi}^{\uparrow^1} \\ z_{\lambda}^{\uparrow^1} \end{pmatrix} :!d_{acc} \vdash \begin{pmatrix} ((more \bullet logicians) \bullet (\epsilon \bullet (met \bullet z_{\Phi}))) \bullet (than \bullet (physicists \bullet (\epsilon \bullet (knew \bullet him)))) \\ \mathbf{More}(\lambda x.\, \mathbf{Logician}(x) \wedge \mathbf{Meet}_{Past}(x, z_{\lambda})\ ,\ \lambda x.\, \mathbf{Physicist}(x) \wedge \mathbf{Know}_{Past}(x, x_{\lambda})) \end{pmatrix} : c; \emptyset}
}{
\begin{pmatrix} y_{\Phi}^{\uparrow^1} \\ y_{\lambda}^{\uparrow^1} \end{pmatrix} :!d_{acc} \vdash \begin{pmatrix} ((more \bullet logicians) \bullet (\epsilon \bullet (met \bullet y_{\Phi}))) \bullet (than \bullet (physicists \bullet (\epsilon \bullet (knew \bullet him)))) \\ \mathbf{More}(\lambda x.\, \mathbf{Logician}(x) \wedge \mathbf{Meet}_{Past}(x, y_{\lambda})\ ,\ \lambda x.\, \mathbf{Physicist}(x) \wedge \mathbf{Know}_{Past}(x, y_{\lambda})) \end{pmatrix} : c; \emptyset
}
$$

The partial derivation above stems from merging the two previously presented branches into one tree. Contraction rule is then applied to encapsulate both controlled hypotheses linked to e_1 in one assumption $y^{\uparrow^1}$. The current history of this latter compound assumption is: hist($y^{\uparrow^1}$)={(1,4) ; (2,5)}.

$$
\dfrac{
\dfrac{}{\vdash \begin{pmatrix} \lambda P_{\Phi}.P_{\Phi}(Godel) \\ \lambda P_{\lambda}.P_{\lambda}(\mathbf{Godel}) \end{pmatrix} : (!d_{acc} \multimap c) \multimap c; \{e_1\}}\, Lex
\qquad
\dfrac{\dots}{\begin{pmatrix} y_{\Phi}^{\uparrow^1} \\ y_{\lambda}^{\uparrow^1} \end{pmatrix} :!d_{acc} \vdash \begin{pmatrix} (more \bullet logicians) \bullet \dots \\ \mathbf{More}(\dots, \dots) \end{pmatrix} : c; \emptyset}
}{
\vdash \begin{pmatrix} ((more \bullet logicians) \bullet (met \bullet Godel)) \bullet (than \bullet (physicists \bullet (knew \bullet him))) \\ \mathbf{More}(\lambda x.\, \mathbf{Logician}(x) \wedge \mathbf{Meet}_{Past}(x, \mathbf{Godel})\ ,\ \lambda x.\, \mathbf{Physicist}(x) \wedge \mathbf{Know}_{Past}(x, \mathbf{Godel})) \end{pmatrix} : c; \emptyset
}\, \multimap IE
$$

The whole derivation ends by simultaneously discharging controlled hypotheses linked to entry e_1 by means of ⊸IE rule. In fact, the application of this rule is allowed since the side-condition ‡ is entirely verified: as $y^{\uparrow^1}$'s history shows, the leftmost hypothesis linked to e_1 was introduced in the derivation after the rightmost one. The semantic representation of our sentence is computed in tandem. Indeed, the final semantics coincides with the intuitive meaning of the sentence, namely that the set of logicians who met Godel is larger than the range of physicists that knew him.

2.4 Enhancing $\mathcal{LGE}$

It is not difficult to notice that our logic is too flexible as the application of movement is not constrained. For instance, if we assign the entry below[7] to

[7] d_{dat} represent noun phrases with dative case

the wh-element '*which*', we can analyze both sentences **which man do you think the child of _ speaks?* and '*which man do you think John loves the child of _?*', where the first is ungrammatical.

$$\vdash \left(\begin{array}{c} \lambda m\ \lambda\phi\ (which \bullet_{<} m) \bullet_{>} \phi(\epsilon) \\ \lambda P\ \lambda Q\ \lambda x.P(x) \wedge Q(x) \end{array} \right) : n \multimap (d_{dat} \multimap c) \multimap c \ \ \text{-}\!\!\circ\!\!\!\!\exists\ \ [X : d_{dat} \vdash X : d_{dat}]$$

In fact, we need to control displacement operation to rule out extraction from islands. For that purpose, we propose to enhance $\mathcal{LGE}$ with some meta-rules encoding locality constraints (e.g. SPIC: Specifier Island Condition, SMC: Shortest Move Condition). We focus in the following on the *SPIC* defined in Koopman and Szabolcsi (2000) which stipulates that the moved element should be a member of the extraction domain (i.e. $comp^+$: transitive closure of the complement relation, or a specifier of a $comp^+$).

In order to locate the position of the head, the complement and the specifier inside a phonetic expression, we decorate the building structure connective $\bullet$ with a mode of composition taken from the set $\{<, >\}$. This mode points towards the sub-tree where the head is located: $\bullet_<$ (resp. $\bullet_>$) if the head is located on the left (resp. right) sub-tree.

A linked lexical entry which is expected to undergo an overt constituent movement has a phonetic-term that obeys the following syntax:

$$\lambda\, x_1 \ldots \lambda\, x_n\ \lambda\, P_\Phi\ \lambda\, y_1 \ldots \lambda\, y_k.\ \mathrm{g}(y_1, \ldots, y_k, \mathrm{f}(x_1, \ldots, x_n) \bullet_{>} P_\Phi(\epsilon))$$

In the expression above, $x_1, \ldots, x_n, y_1, \ldots, y_k$ ($n \geq 0$, $k \geq 0$) are Φ-variables of arbitrary types, whereas P_Φ is a Φ-variable of type $s \multimap s$. Moreover, f (resp. g) is a function that takes n (resp. k+1) Φ-terms and builds a phonetic structure using these parameters together with constants of Σ.

Intuitively, this syntax means that our entry will firstly combine with n structures $x_1, \ldots, x_n$ by means of merge operation, thus leading to a maximal projection $f(x_1, \ldots, x_n)$. Then, the intermediary sites will be replaced by traces in the initial structure P_Φ and our maximal projection will be placed in specifier position, hence making it possible to carry out the expected movement. Finally, our resulting constituent can merge with other structures, thus yielding a complete expression (e.g. *whom* entry in section 2.2).

Notice that this syntax suits the type specification defined in section 2.2 (points 2 & 3) if we add additional conditions, namely that both types t (type of intermediary sites) and t' (type of the D-structure before movement) are atomic. The first condition (i.e. $t \in \mathcal{A}$) follows from constraints proposed by Koopman and Szabolcsi Koopman and Szabolcsi (2000) which forces moved elements to be maximal projections (i.e. complete expressions). However, the latter condition ($t' \in \mathcal{A}$) is a logical formalization of the *merge over move* principle Chomsky (1995) which stipulates that merge operation has priority over movement because of its simplicity. Therefore, a structure that will undergo move operation should be complete.

According to the syntax of phonetic terms associated with moved elements, SPIC condition can be encoded in $\mathcal{LGE}$ as a pre-condition of $\multimap$IE rule (cf. Fig 3) stipulating the inclusion of all occurrences of Φ-variable c_Φ within the extraction domain of the Φ-term d_Φ. Therefore, adding this meta-rule to $\mathcal{LGE}$ prevents us from analyzing the previous ungrammatical sentence.

2.5 Conclusion & Future Work

$\mathcal{LGE}$ is a new logical formalism which proposes a deductive simulation of Minimalist Program. Our proposal is powerful enough to describe several linguistic phenomena such as medial extraction, binding, ellipsis and discontinuity thanks to using linked lexical entries (related to controlled hypotheses). Moreover, one can solve over-generation problems caused by the freedom of displacement by adding some meta-rules encoding locality constraints.
In addition, it is not difficult to show that these grammars are richer than context free grammars as they are able to generate crossed-dependencies languages (e.g. $\{a^n b^m c^n d^m \mid \text{n, m} \geq 0\}$). In fact, this latter language is recognized by $\mathcal{LGE}$ grammar containing the lexicon below [8]:

$\vdash \epsilon$: p_i (∀ i∈{1..4})
$\vdash \lambda$x. λy. λz. λu. x•(y•(z•u)): ty
$\vdash \lambda$ P. λx. λy. λz. λu. P(a•x, y, c•z, u): ty $\multimap$ ty
$\vdash \lambda$ P. λx. λy. λz. λu. P(x, b•y, z, d•u): ty $\multimap$ ty

The next direction to explore concerns the study of $\mathcal{LGE}$ formal properties: expressive power, decidability, and complexity. We also intend to build bridges between $\mathcal{LGE}$ and other well-known grammatical frameworks (e.g. Minimalist Grammar, TAGs).
Finally, we are developing a meta-linguistic toolkit using **Coq** proof assistant (Coq Team (2004)), in order to study logical properties of $\mathcal{LGE}$ grammars being enhanced with packages of meta-constraints. This toolkit can help users manage complex derivations by automatically handling some technical proofs thanks to powerful computation tools (strategies).

References

Chomsky, N. 1995. *The minimalist program*. MIT Press.

Coq Team. 2004. The coq proof assistant, reference manual, version 8.0. Tech. rep., INRIA.

Curry, HB. 1961. Some logical aspects of grammatical structures. In R. Jackobson, ed., *Symposium in Applied Mathematics*, pages 56–68.

[8] In that case, $\mathcal{A}$={p_1,p_2,p_3,p_4,c} and *ty* denotes the composite type $p_1 \multimap p_2 \multimap p_3 \multimap p_4 \multimap$c

de Groote, P. 2001. Towards abstract categorial grammars. In *39th Annual Meeting of the Association for Computational Linguistics*. Toulouse.

Girard, Jean-Yves. 1987. Linear logic. *Theoretical Computer Science* 50:1–102.

Harkema, H. 2000. A recognizer for minimalist grammars. In *IWPT*.

Hendriks, P. 1995. *Comparatives and Categorial Grammar*. Ph.D. thesis, University of Groningen, The Netherlands.

Kayne, R. 2002. Pronouns and their antecedents. In S. E. . T. Seely, ed., *Derivation and Explanation in the Minimalist Program*. Blackwell.

Koopman, H. and A. Szabolcsi. 2000. Verbal complexes. In *Current series in Linguistic Theory*. MIT Press.

Lambek, J. 1958. The mathematics of sentence structure. *American Mathematical Monthly* .

Lecomte, A and C Retore. 2001. Extending lambek grammars: a logical account of minimalist grammars. In *39th Annual Meeting of the Association for Computational Linguistics*, pages 354–362. Toulouse.

Moortgat, M. 1997. Categorial type logic. In V. B. . ter Meulen, ed., *Handbook of Logic and Language*, chap. 2. Elsevier.

Morrill, G. 2000. Type logical anaphora. Tech. rep., Universitat Politecnica, Catalunya.

Muskens, R. 2003. Language, lambdas, and logic. In *Resource Sensitivity in Binding and Anaphora*, Studies in Linguistics and Philosophy. Kluwer.

Stabler, E. 1997. Derivational minimalism. In C. Retore, ed., *Logical Aspects of Computational Linguistics*. Springer.

Vermaat, W. 1999. *Controlling Movement: Minimalism in a deductive perspective*. Master's thesis, master's thesis, Utrecht University.

3

P-TIME Decidability of NL1 with Assumptions

MARIA BULIŃSKA

Abstract

Buszkowski (2005) showed that systems of Nonassociative Lambek Calculus with finitely many nonlogical axioms are decidable in polynomial time and generate context-free languages. The same holds for systems with unary modalities, studied in Moortgat (1997), *n*-ary operations, and the rule of permutation, studied in Jäger (2004). The polynomial time decidability for Classical Nonassociative Lambek Calculus was established by de Groote and Lamarche (2002). We study Nonassociative Lambek Calculus with identity enriched with a finite set of assumptions. To prove that this system is decidable in polynomial time we adapt the method used in Buszkowski (2005). The modification is essential. The novelty is the lemma about the elimination of cut rules with premisses with empty antecedents for some auxiliary system. The context-freeness of the languages generated of the systems of Nonassociative Lambek Calculus is also established.

Keywords LAMBEK CALCULUS, P-TIME DECIDABILITY

3.1 Introduction and preliminaries

Nonlogical axioms can be of interest for linguistics for several reason. We can use them to describe subcategorization in natural language. For instance, restrictive adjectives are a subcategory of adjectives. Further, by enriching Nonassociative Lambek Calculus with finitely new axioms, we can improve its expressibility without lacking the nice computational simplicity.

First we describe the formalism of Nonassociative Lambek Calculus with identity constant (NL1). Let At = {p, q, r, . . .} be the denumerable set of atoms (primitive types).

The set of formulas (also called types) Tp1 is defined as the smallest set fulfilling the following conditions:

FG-2006.
Edited by Paola Monachesi, Gerald Penn,
Giorgio Satta and Shuly Wintner.

- $\mathbf{1} \in$ Tp1,
- At $\subseteq$ Tp1,
- if $A, B \in$ Tp1, then $(A \bullet B) \in$ Tp1, $(A/B) \in$ Tp1, $(A \backslash B) \in$ Tp1, where binary connectives $\backslash$, $/$, $\bullet$, are called *left residuation*, *right residuation*, and *product*, respectively.

The set of formula structures STR1 is defined recursively as follows:

- $\Lambda \in$ STR1, where Λ denotes an empty structure,
- Tp1 $\subseteq$ STR1; these formula structures are called atomic formula structures,
- if $X, Y \in$ STR1, then $(X \circ Y) \in$ STR1.

We set $(X \circ \Lambda) = (\Lambda \circ X) = X$.
Substructures of a formula structure are defined in the following way:

- Λ is only substructure of Λ,
- if X is an atomic formula structure, then Λ and X are only substructures of X,
- if $X = (X_1 \circ X_2)$, then X and all substructures of X_1 and X_2 are substructures of X.

By $X[Y]$ we denote a formula structure X with a distinguished substructure Y, and by $X[Z]$ - the substitution of Z for Y in X.
Sequents are formal expressions $X \to A$ such that $A \in$ Tp1, $X \in$ STR1.

The Gentzen-style axiomatization of the calculus NL1 employs the axiom schemas:

$$(\mathrm{Id}) \quad A \to A \qquad\qquad (\mathbf{1}\mathrm{R}) \quad \Lambda \to \mathbf{1}$$

and the following rules of inference:

$$(\mathbf{1}\mathrm{L}) \quad \frac{X[\Lambda] \to A}{X[\mathbf{1}] \to A},$$

$$(\bullet\mathrm{L}) \quad \frac{X[A \circ B] \to C}{X[A \bullet B] \to C}, \qquad (\bullet\mathrm{R}) \quad \frac{X \to A; \quad Y \to B}{X \circ Y \to A \bullet B},$$

$$(\backslash\mathrm{L}) \quad \frac{Y \to A; \quad X[B] \to C}{X[Y \circ (A \backslash B)] \to C}, \qquad (\backslash\mathrm{R}) \quad \frac{A \circ X \to B}{X \to A \backslash B},$$

$$(/\mathrm{L}) \quad \frac{X[A] \to C; \quad Y \to B}{X[(B/A) \circ Y] \to C}, \qquad (/\mathrm{R}) \quad \frac{X \circ B \to A}{X \to A/B},$$

$$(\mathrm{CUT}) \quad \frac{Y \to A; \quad X[A] \to B}{X[Y] \to B}.$$

For any system S we write S $\vdash X \to A$ if the sequent $X \to A$ is derivable in S.

The most general models of NL1 are residuated groupoid with identity.
A *residuated groupoid* with identity is a structure

$$\mathcal{M} = (M, \leq, \cdot, \backslash, /, 1)$$

such that

- $(M, \cdot, 1)$ is a groupoid with identity in which $a \cdot 1 = a$, $1 \cdot a = a$ for all $a \in M$
- $(M, \leq)$ is a poset ,
- $\backslash, /$ are binary operations on M satisfying the equivalences :

$$\text{(RES)} \quad ab \leq c \quad \text{iff} \quad b \leq a\backslash c \quad \text{iff} \quad a \leq c/b$$

for all $a, b, c \in M$.

Every residuated groupoid fulfills the following monotonicity laws:

$$\text{(MON)} \quad \text{If} \quad a \leq b \quad \text{then} \quad ca \leq cb \quad \text{and} \quad ac \leq bc$$

$$\text{(MRE)} \quad \text{If} \quad a \leq b \quad \text{then} \quad c\backslash a \leq c\backslash b, \quad a/c \leq b/c,$$
$$b\backslash c \leq a\backslash c, \quad c/b \leq c/$$

for all $a, b, c \in M$.

A *model* is a pair $(\mathcal{M}, \mu)$ such that $\mathcal{M}$ is a residuated groupoid with identity and μ is an assignment of elements of M for atoms. One extends μ for all formulas :

$$\mu(\mathbf{1}) = 1, \quad \mu(A \bullet B) = \mu(A) \cdot \mu(B),$$
$$\mu(A\backslash B) = \mu(A)\backslash\mu(B), \quad \mu(A/B) = \mu(A)/\mu(B).$$

and formula structure:

$$\mu(\Lambda) = \mu(\mathbf{1}) = 1, \quad \mu(X \circ Y) = \mu(X) \cdot \mu(Y).$$

A sequent $X \to A$ is said to be true in model $(\mathcal{M}, \mu)$ if $\mu(X) \leq \mu(A)$. In particular a sequent $\Lambda \to A$ is said to be true in model $(\mathcal{M}, \mu)$ if $1 \leq \mu(A)$.
One can prove the following property for formula structures:

$$\text{(MON} - \text{STR)} \quad \text{If} \quad \mu(Y) \leq \mu(Z) \quad \text{then} \quad \mu(X[Y]) \leq \mu(X[Z]).$$

3.2 NL1 with assumptions

Let Γ be a set of sequents of the form $A \to B$, where $A, B \in$ Tp1. By NL1(Γ) we denote the calculus NL1 with additional set Γ of assumptions. NL1 is strongly complete with respect to the residuated groupoids with identity, i.e. all sequents provable in NL1(Γ) are precisely those which are true in all models $(\mathcal{M}, \mu)$ in which all sequents from Γ are true. Soundness is easily proved by induction on derivation in NL1(Γ). Completeness follows from the fact that the Lindenbaum algebra of NL1 is a residuated groupoid with identity.

In general, the calculus NL1(Γ) has not the standard subformula property, since (CUT) is legal rule in this system . Thus we take into consideration the subformula property in some extended form.
Let T be a set of formulas closed under subformulas and such that all formulas appearing in Γ belong to T. By a T-sequent we mean a sequent $X \to A$ such that A and all formulas appearing in X belong to T. Now, we can reformulate the subformula property as follows:

> Every T-sequent provable in a system S has a proof in S such that all sequents appearing in this proof are T-sequents.

To prove the subformula property for NL1(Γ) we will use special models, namely a residuated groupoids with identity of cones over given preordered groupoids with identity.
Let $(M, \leq, \cdot)$ be a preordered groupoid, that means, it is a groupoid with a preordering (i.e. a reflexive and transitive relation), satisfying (MON).
A set $P \subseteq M$ is called a *cone* on M if $a \leq b$ and $b \in P$ entails $a \in P$. Let $C(M)$ denotes the set of cones on M.
The operations $\cdot, \backslash, /$ on $C(M)$ are defined as follows:

$$\text{(M1)} \quad I = \{a \in M : a \leq 1\}$$

$$\text{(M2)} \quad P_1 P_2 = \{c \in M : (\exists a \in P_1, b \in P_2)\ c \leq ab\}$$

$$\text{(M3)} \quad P_1 \backslash P_2 = \{c \in M : (\forall a \in P_1)\ ac \in P_2\}$$

$$\text{(M4)} \quad P_1 / P_2 = \{c \in M : (\forall b \in P_2)\ cb \in P_1\}.$$

A structure $(C(M), \subseteq, \cdot, \backslash, /, I)$ is a residuated groupoid with identity. It is called the residuated groupoid with identity of cones over the given preordered groupoid with identity.

Let M be the set of all formula structures all of whose atomic substructures belong to T and $\Lambda \in M$. If a sequent $X \to A$ has a proof in NL1(Γ) consisting of T-sequents only, we write: $X \to_T A$.
First, we define on M a relation $\leq_b$. $X \leq_b Y$ denotes X directly reduces to Y. The definition of this relation is as follows:

$$Y[Z] \leq_b Y[\Lambda] \quad \text{if} \quad Z \to_T \mathbf{1},$$

$$Y[Z] \leq_b Y[A] \quad \text{if} \quad Z \to_T A,$$

$$Y[A \bullet B] \leq_b Y[A \circ B] \quad \text{if} \quad A \bullet B \in T.$$

A preordering $\leq$ on M is defined as a reflexive and transitive closure of the relation $\leq_b$. Then $X \leq Y$ iff there exist $Y_0, \ldots, Y_n, n \geq 0$ such that $X = Y_0, Y = Y_n$ and $Y_{i-1} \leq_b Y_i$, for each $i = 1, \ldots, n$.
Clearly, $(M, \leq, \circ, \Lambda)$ is a preordered groupoid with identity Λ fulfilling (MON).

Next, we take into consideration the residuated groupoid of cones with identity $C(M) = (C(M), \subseteq, \cdot, \backslash, /, I)$ over $(M, \leq, \circ, \Lambda)$ defined above. An assignment μ on $C(M)$ is defined by setting:

$$\mu(p) = \{X \in M : X \rightarrow_T p\},$$

for all atoms p. One can easily prove that

$$\mu(A) = \{X \in M : X \rightarrow_T A\},$$

for all $A \in T$.

Fact 1 *Every sequent provable in* NL1(Γ) *is true in* $(C(M), \mu)$.

Proof. It suffice to show, that each axiom from Γ is true in $(C(M), \mu)$. Assume that $A \rightarrow B$ belongs to Γ. It yields $A \rightarrow_T B$. We need to show that $\mu(A) \subseteq \mu(B)$. Let $X \in \mu(A)$. Then, $X \rightarrow_T A$. By (CUT), we get $X \rightarrow_T B$, which yields $X \in \mu(B)$. □

Lemma 2 *The system* NL1(Γ) *has the extended subformula property.*

Proof. Let $X \rightarrow A$ be a T-sequent provable in NL1(Γ). By fact 1 it is true in the model (C, μ), i.e. $\mu(X) \subseteq \mu(A)$. Since $X \in \mu(X)$, we have $X \in \mu(A)$. But it means $X \rightarrow_T A$. □

A sequent is said to be *basic* if it is a T-sequent of the form $\Lambda \rightarrow A$, $A \rightarrow B$, $A \circ B \rightarrow C$. Let Γ be finite, and let T be a finite set of formulas, closed under subformulas and such that T contains all formulas appearing in Γ. For such T we shall describe an effective procedure which produces all basic sequents derivable in NL1(Γ).

Let S_0 consist of all T-sequent of the form (Id), all sequents from Γ, $\Lambda \rightarrow \mathbf{1}$ and all T-sequents of the form:

$$\mathbf{1} \circ A \rightarrow A,\ A \circ \mathbf{1} \rightarrow A,\ A \circ B \rightarrow A \bullet B,$$
$$A \circ (A \backslash B) \rightarrow B,\ (A/B) \circ B \rightarrow A.$$

Assume S_n has already been defined. S_{n+1} is S_n enriched with sequents resulting from the following rules:

(S1) if $(A \circ B \rightarrow C) \in S_n$ and $(A \bullet B) \in T$, then $(A \bullet B \rightarrow C) \in S_{n+1}$,
(S2) if $(A \circ X \rightarrow C) \in S_n$ and $(A \backslash C) \in T$, then $(X \rightarrow A \backslash C) \in S_{n+1}$,
(S3) if $(X \circ B \rightarrow C) \in S_n$ and $(C/B) \in T$, then $(X \rightarrow C/B) \in S_{n+1}$,
(S4) if $(\Lambda \rightarrow A) \in S_n$ and $(A \circ X \rightarrow C) \in S_n$, then $(X \rightarrow C) \in S_{n+1}$,
(S5) if $(\Lambda \rightarrow A) \in S_n$ and $(X \circ A \rightarrow C) \in S_n$, then $(X \rightarrow C) \in S_{n+1}$,
(S6) if $(A \rightarrow B) \in S_n$ and $(B \circ X \rightarrow C) \in S_n$, then $(A \circ X \rightarrow C) \in S_{n+1}$,
(S7) if $(A \rightarrow B) \in S_n$ and $(X \circ B \rightarrow C) \in S_n$, then $(X \circ A \rightarrow C) \in S_{n+1}$,
(S8) if $(A \circ B \rightarrow C) \in S_n$ and $(C \rightarrow D) \in S_n$, then $(A \circ B \rightarrow D) \in S_{n+1}$.

Clearly, $S_n \subseteq S_{n+1}$ for all $n \geq 0$. We define S^T as the join of this chain. S^T is a set of basic sequents, hence it must be finite. It yields $S^T = S_{k+1}$, for the least k such that $S_k = S_{k+1}$, and this k is not greater then the number of basic sequents.

Fact 3 *The set S^T can be constructed in polynomial time.*

Proof. Let n be the cardinality of T. There are n, n^2 and n^3 basic sequents of the form $\Lambda \to A$, $A \to B$ and $A \circ B \to C$, respectively. Hence, we have $m = n^3 + n^2 + n$ basic sequents. The set S_0 can be constructed in time $0(n^2)$. To get S_{i+1} from S_i we must close S_i under the rules (S1)-(S8) which can be done in at most m^3 steps for each rule. For example, to close S_i under (S1) we must check if $(A \circ B \to C) \in S_i$ and $(A \bullet B) \in T$ which needs at most m and n steps, respectively. The sequent $A \bullet B \to C$ is added to S_{i+1} only if it doesn't belong to this set. To check this fact the next m steps are needed. The least k such that $S^T = S_k$ is at most m. Then finely, we can construct S^T from T in time $0(m^4) = 0(n^{12})$. □

By $S(T)$ we denote the system whose axioms are all sequents from S^T and whose only inference rule is (CUT). Then, every proof in $S(T)$ consist of T-sequents only.
The fact that every basic sequent provable in $S(T)$ belongs to S^T, which is used in a proof of an interpolation lemma for $S(T)$, is not obvious in NL1(Γ), because of sequents of the form $\Lambda \to A$.
By $S(T)^-$ we denote the system whose axioms are all sequents from S^T and whose only inference rule is (CUT) with premises without empty antecedents.

Lemma 4 *For any sequent $X \to A$, $S(T) \vdash X \to A$ iff $S(T)^- \vdash X \to A$.*

Proof. The 'if' direction is evident. To prove the 'only if' direction we show that $S(T)^-$ is closed under (CUT), i.e.

(*) If $S(T)^- \vdash X \to B$ and $S(T)^- \vdash Y[B] \to A$, then $S(T)^- \vdash Y[X] \to A$.

Assume $S(T)^- \vdash X \to B$ and $S(T)^- \vdash Y[B] \to A$.
If $X \neq \Lambda$, then $S(T)^- \vdash Y[X] \to A$ by definition of $S(T)^-$.
If $X = \Lambda$, then the sequent $X \to B$ is of the form $\Lambda \to B$ and $S(T)^- \vdash \Lambda \to B$, which means that $\Lambda \to B$ is an axiom of $S(T)^-$. To prove (*) we proceed by induction on derivation of second premise: $Y[B] \to A$.
If $Y[B] \to A$ is an axiom of $S(T)^-$, then $(Y[B] \to A) \in S^T$. S^T is closed under (CUT). Hence, $(Y[\Lambda] \to A) \in S^T$ which yields $S(T)^- \vdash Y[\Lambda] \to A$.
If $Y[B] \to A$ is a conclusion of (CUT) from premises without empty antecedents, then $Y[B] = Z[Y']$ and for some $C \in T$, $S(T)^- \vdash Y' \to C$ and $S(T)^- \vdash Z[C] \to A$. We consider the following cases.

I. B is contained in Y'. Then $Y' = Y'[B]$.

(1) $Y'[B] \neq B$. By the induction hypothesis, (*) holds for $\Lambda \to B$ and $Y'[B] \to C$, so $S(T)^- \vdash Y'[\Lambda] \to C$. Since $Y'[B] \neq B$, we have $Y'[\Lambda] \neq \Lambda$. Using (CUT), we get $S(T)^- \vdash Z[Y'[\Lambda]] \to A$, which means $S(T)^- \vdash$

$Y[\Lambda] \to A$.

(2) $Y'[B] = B$. By the induction hypothesis, (*) holds for $\Lambda \to B$ and $B \to C$, so $S(T)^- \vdash \Lambda \to C$. Using inductive hypothesis to $\Lambda \to C$ and $Z[C] \to A$, we get $S(T)^- \vdash Z[\Lambda] \to A$, which means $S(T)^- \vdash Y[\Lambda] \to A$.

II. B and Y' do not overlap. Then B is contained in Z and does not overlap C in Z. We write $Z[C] = Z[B, C]$. From the assumption we have $Y' \neq \Lambda$. By induction hypothesis, (*) holds for $\Lambda \to B$ and $Z[B, C] \to A$, so $S(T)^- \vdash Z[\Lambda, C] \to A$. By (CUT), $S(T)^- \vdash Z[\Lambda, Y'] \to A$, which means $S(T)^- \vdash Y[\Lambda] \to A$.

□

Corollary 5 *Every basic sequents provable in $S(T)$ belongs to S^T.*

Proof. We proceed by induction on proofs in $S(T)$. Assume $X \to A$ is a basic sequent derivable in $S(T)$. If $X \to A$ is an axiom of $S(T)$, then $(X \to A) \in S^T$. If $X \to A$ is a conclusion of (CUT), we consider three cases.

(1) $X = \Lambda$. By lemma 4, $\Lambda \to A$ has a proof in $S(T)^-$. Hence $\Lambda \to A$ is an axiom, which means $(\Lambda \to A) \in S^T$.

(2) $X = B$. By lemma 4, there exists a proof such that $B \to A$ is a conclusion from premises $B \to C$ and $C \to A$, where $C \neq \Lambda$. Since proofs in S(T) consist with T-sequents only, $B \to C$ and $C \to A$ are basic sequents. By induction hypothesis, $(B \to C) \in S^T$ and $(C \to A) \in S^T$. S^T is closed under (CUT), so $(B \to A) \in S^T$.

(3) $X = B \circ C$. By lemma 4, there exists a proof such that $B \circ C \to A$ is a conclusion from premises without empty premises. Hence, they are of the form: $(B \circ C \to D, D \to A)$ or $(B \to D, D \circ C \to A)$ or $(C \to D, B \circ D \to A)$. By the same argument as in (2), in each case, we get $(B \circ C \to A) \in S^T$.

□

Now, we can state an interpolation lemma for $S(T)$.

Lemma 6 *If $S(T) \vdash X[Y] \to A$, then there exists $D \in T$ such that $S(T) \vdash Y \to D$ and $S(T) \vdash X[D] \to A$.*

Proof. We proceed by induction on proofs in $S(T)$.

I. Assume $X[Y] \to A$ is an axiom of $S(T)$. We consider the following cases.

(1) $X[Y] = Y$. Then $Y = X$ (observe, that this case includes subcase $X = \Lambda$). We set $D = A$. We have $S(T) \vdash X \to A$ from assumption and $S(T) \vdash A \to A$, since $(A \to A) \in S^T$.

(2) $X[Y] = B$, $Y = \Lambda$. Then $X[Y] = X[\Lambda] = B = B \circ \Lambda$ or $X[Y] = \Lambda \circ B$ and $D = \mathbf{1}$. We have $S(T) \vdash \Lambda \to \mathbf{1}$ and $S(T) \vdash B \to A$. $(B \circ \mathbf{1} \to B) \in S^T$,

so $S(T) \vdash B \circ \mathbf{1} \to B$. Using (CUT) we get $S(T) \vdash X[\mathbf{1}] \to A$. For $X[Y] = \Lambda \circ B$ the argument is dual.

(3) $X[Y] = B \circ C$, $Y \neq \Lambda$. Then $Y = B$ or $Y = C$, hence $D = B$ or $D = C$, respectively.

(4) $X[Y] = B \circ C$, $Y = \Lambda$. Then $X[\Lambda]$ has one of the form: $\Lambda \circ (B \circ C)$, $(B \circ C) \circ \Lambda$, $(\Lambda \circ B) \circ C$, $(B \circ \Lambda) \circ C$, $B \circ (\Lambda \circ C)$, $B \circ (C \circ \Lambda)$. For example, if $X[\Lambda] = \Lambda \circ (B \circ C)$, we have $S(T) \vdash \Lambda \to \mathbf{1}$ and using (CUT) to $S(T) \vdash B \circ C \to A$ and $S(T) \vdash \mathbf{1} \circ A \to A$, we get $S(T) \vdash \mathbf{1} \circ (B \circ C) \to A$.

II. Assume $X[Y] \to A$ is the conclusion of (CUT). Then $X[Y] = Z[Y']$ and for some $B \in T$: $S(T) \vdash Y' \to B$ and $S(T) \vdash Z[B] \to A$.

In this part the proof is analogous to the proof of lemma 2 in Buszkowski (2005). The following cases are considered.

(1) Y is contained in Y'. Then $Y' = Y'[Y]$. By the induction hypothesis, there exists $D \in T$ such that $S(T) \vdash Y \to D$ and $S(T) \vdash Y'[D] \to B$. Using (CUT) with the premises $Z[B] \to A$ and $Y'[D] \to B$ we get $S(T) \vdash Z[Y'[D]] \to A$, which means $S(T) \vdash X[D] \to A$.

(2) Y' is contained in Y. Then $X[Y] = X[Y[Y']] = Z[Y']$ and $Z[B] = X[Y[B]]$. By the induction hypothesis, there exists $D \in T$ such that $S(T) \vdash Y[B] \to D$ and $S(T) \vdash X[D] \to A$. Using (CUT) with the premises $Y' \to B$ and $Y[B] \to D$ we get $S(T) \vdash Y[Y']] \to D$.

(3) Y and Y' do not overlap. Then Y is contained in Z and does not overlap B in Z. We write $Z[B] = Z[B, Y]$. By the induction hypothesis, there exists $D \in T$ such that $S(T) \vdash Y \to D$ and $S(T) \vdash Z[B, D] \to A$. Using (CUT) with the premises $Y' \to B$ and $Z[B, D] \to B$ we get $S(T) \vdash Z[Y', D] \to A$, which means $S(T) \vdash X[D] \to A$.

□

Lemma 7 *For any T-sequent $X \to A$, $X \to_T A$ iff $S(T) \vdash X \to A$.*

Proof. Recall, that $X \to_T A$ means that the sequent $X \to A$ has the proof in NL1(Γ) consisting with T-sequents only.
To prove 'if' direction observe that $X \to_T A$, for all sequents $X \to A$ in S^T. The T-sequents which are axioms of NL1(Γ) belong to S_0. Thus, to prove the 'only if' direction it suffices to show that all inference rules of NL1(Γ), restricted to T-sequents, are admissible in $S(T)$. For example, let us consider ($\mathbf{1}$L). Assume $X[\Lambda] \to A$. By lemma 6, there exist $D \in T$ such that $S(T) \vdash \Lambda \to D$ and $S(T) \vdash X[D] \to A$. Since $(D \circ \mathbf{1} \to D) \in S^T$, then $S(T) \vdash D \circ \mathbf{1} \to D$. By two applications of (CUT), we get $S(T) \vdash X[\Lambda \circ \mathbf{1}] \to A$, which means $S(T) \vdash X[\mathbf{1}] \to A$. □

Theorem 8 *If Γ is finite, then* NL1(Γ) *is decidable in polynomial time.*

Proof. Let Γ be a finite set of sequents of the form $B \to C$ and let $X \to A$ be a sequent. Let n be the number of logical constants and atoms in $X \to A$ and Γ.

As T we choose the set of all subformulas of formulas appearing in $X \to A$ and formulas appearing in Γ. Since the number of subformulas of any formula B is equal to the number of logical constants and atoms in B, T has n elements and we can construct it in time $0(n^2)$. By lemma 2, $\mathrm{NL1}(\Gamma) \vdash X \to A$ iff $X \to_T A$. By lemma 7, $X \to_T A$ iff $S(T) \vdash X \to A$. Proofs in $S(T)$ are actually derivation trees of a context-free grammar whose production rules are the reversed sequents from S^T. Checking derivability in context-free grammars is P-TIME decidable. For example, by known CYK algorithm, it can be done in time not exceed $k \cdot n^3$, where k is the size of S^T. By the proof of fact 3, the size of S^T is at most $0(n^3)$ and S^T can be constructed in $0(n^{12})$. Hence, the total time is $0(n^{12})$, i.e. NL1(Γ) is P-TIME decidable. □

By theorem 8, we have immediately that languages generated by the categorial grammar based on the system NL1(Γ) are context-free. In Buszkowski (2005) the analogous result was established for NL(Γ), NL(Γ) with permutation rule and Generalized Lambek Calculus (GLC(Γ)). The context-freeness of the languages generated by Nonassociative Lambek Calculus were studied by Buszkowski (1986), Kandulski (1988) and Jäger (2004). Bulińska (2005) obtained the weak equivalence of context-free grammars and grammars based on the associative Lambek calculus with finite set of simple nonlogical axioms of the form $p \to q$, where p, q are primitive types.

References

Bulińska, M. 2005. The Pentus Theorem for Lambek Calculus with Simple Nonlogical Axioms. *Studia Logica* 81:43–59.

Buszkowski, W. 1986. Generative capacity of Nonassociative Lambek Calculus. *Bulletin of Polish Academy of Sciences. Mathematics* 34:507–516.

Buszkowski, W. 2005. Lambek Calculus with Nonlogical Axioms. In C. Casadio, P. J. Scott, and R. A. G. Seely, eds., *Language and Grammar*, Studies in Mathematical Linguistics and Natural Language, pages 77–93. CSLI Publications.

de Groote, P. and F. Lamarche. 2002. Clasical Non-Associative Lambek Calculus . *Studia Logica* 71:355–388. Special issue: The Lambek calculus in logic and linguistics.

Jäger, G. 2004. Residuation, Structural Rules and Context Freeness. *Journal of Logic, Language and Information* 13:47–59.

Kandulski, M. 1988. The equivalence of Nonassociative Lambek Categorial Grammars and Context-Free Grammars. *Zeitschrift für mathematische Logik und Grundlagen der Mathematik* 52:34–41.

Moortgat, M. 1997. Categorial Type Logics. In J. van Benthem and A. ter Meulen, eds., *Handbook of Logic and Language*, pages 93–177. Amsterdam, Cambrigde Mass.: Elsvevier, MIT Press.

4

Program Transformations for Optimization of Parsing Algorithms and Other Weighted Logic Programs

Jason Eisner and John Blatz

Abstract

Dynamic programming algorithms in statistical natural language processing can be easily described as weighted logic programs. We give a notation and semantics for such programs. We then describe several source-to-source transformations that affect a program's efficiency, primarily by rearranging computations for better reuse or by changing the search strategy.

Keywords weighted logic programming, dynamic programming, program transformation, parsing algorithms

4.1 Introduction

In this paper, we show how some efficiency tricks used in the natural language processing (NLP) community, particularly for parsing, can be regarded as specific instances of transformations on weighted logic programming algorithms.

We define weighted logic programs and sketch the general form of the transformations, enabling their application to new programs in NLP and other domains. Several of the transformations (folding, unfolding, magic templates) have been known in the logic programming community, but are generalized here to our weighted framework and applied to NLP algorithms. We also present a powerful generalization of folding—speculation—which appears new and is able to derive some important parsing algorithms. Finally, our formalization of these transformations has been simplified by our use of "gap

FG-2006.
Edited by Paola Monachesi, Gerald Penn,
Giorgio Satta and Shuly Wintner.

passing" ideas from categorial grammar and non-ground terms from logic programming.

The framework that we use for specifying the weighted logic programs is roughly based on that of Dyna (Eisner et al., 2005), an implemented system that can compile such specifications into efficient C++. Some of the programs could also be handled by PRISM (Zhou and Sato, 2003), an implemented probabilistic Prolog.

It is especially useful to have general optimization techniques for dynamic programming algorithms (a special case in our framework), because NLP researchers regularly propose new such algorithms. Dynamic programming is used to parse many different grammar formalisms. It is also used in stack decoding, grammar induction, finite-state methods, and syntax-based approaches to machine translation and language modeling.

One might select program transformations either manually or automatically. Our goal here is simply to illustrate the search space of semantically equivalent programs. We do not address the practical question of searching this space—that is, the question of where and when to apply the transformations. For some programs and their typical inputs, a transformation will speed a program up; in other cases, it will slow it down. The actual effect can of course be determined empirically by running the transformed program (or in some cases, predicted more quickly by profiling the *untransformed* program as it runs on typical inputs). Thus, at least in principle, one could apply automatic local search methods.

4.2 Our Formalism

4.2.1 Logical Specification of Dynamic Programs

We will use context-free parsing as a simple running example. Recall that one can write a logic program for CKY recognition (Younger, 1967) as follows, where constit(X,I,K) is provable iff the grammar, starting at nonterminal X, can generate the input substring from position I to position K.

```
constit(X,I,K) :- rewrite(X,W), word(W,I,K).
constit(X,I,K) :- rewrite(X,Y,Z), constit(Y,I,J), constit(Z,J,K).
goal :- constit(s,0,N), length(N).

rewrite(s,np,vp).          % tiny grammar
rewrite(np,"Dumbo").
rewrite(np,"flies").
rewrite(vp,"flies").

word("Dumbo",0,1).     % tiny input sentence
word("flies",1,2).
length(2).
```

We say that this logic program is a **dynamic program** because it satisfies a simple restriction: all **variables** (capitalized) in a rule's left-hand side (rule **head**) also appear on its right-hand side (rule **body**). Logic programs restricted in this way correspond to the "grammatical deduction systems" discussed by Shieber et al. (1995). They can be evaluated by a simple agenda-based, bottom-up dynamic programming algorithm.[1]

This paper, however, deals with general logic programs without this restriction. For example, one may wish to assert the availability of an "epsilon" word at *every* position K in the sentence: word(epsilon,K,K). We emphasize this because it is convenient for some of our transformations to introduce new non-dynamic rules. One can often eliminate non-dynamic rules (in particular, the ones we introduce) to obtain a semantically equivalent dynamic program, but we do not here explore transformations for doing so systematically.

4.2.2 Weighted Logic Programs

We now define our notion of *weighted* logic programs, of which the most useful in NLP are the semiring-weighted dynamic programs discussed by Goodman (1999) and Eisner et al. (2005). See the latter paper for a discussion of relevant work on deductive databases with aggregation (e.g., Fitting, 2002, Van Gelder, 1992, Ross and Sagiv, 1992).

Our running example is the inside algorithm for context-free parsing:

```
constit(X,I,K) += rewrite(X,W) * word(W,I,K).
constit(X,I,K) += rewrite(X,Y,Z) * constit(Y,I,J) * constit(Z,J,K).
goal += constit(s,0,N) * length(N).

rewrite(s,np,vp) = 1.           % p(s → np vp | s)
rewrite(np,"Dumbo") = 0.6.  % p(np → "Dumbo" | np)
rewrite(np,"flies") = 0.4.      % p(vp → "flies" | vp)
rewrite(vp,"flies") = 1.        % p(vp → "flies" | vp)

word("Dumbo",0,1) = 1.      % 1 for all words in the sentence
word("flies",1,2) = 1.
length(2) = 1.
```

This looks just like the unweighted logic program in section 4.2.1, except that now the body of each rule is an arbitrary *expression*, and the :- operator is replaced by an "aggregation operator" such as += or max=. Since line 2 can be instantiated for example as constit(s,0,2) += rewrite(s,np,vp) * constit(np,0,1) * constit(vp,1,2), the value of rewrite(s,np,vp) * constit(np,0,1) * constit(vp,1,2) (if any) is used as a summand (i.e., an operand of +=) in the value of constit(s,0,2).

[1]This is superior to a Prolog-style backtracking algorithm. It runs in polynomial time, rather than wasting exponential time re-deriving the same constituents in different contexts, or failing to terminate if the grammar is left-recursive.

We will formalize this in section 4.2.3 below.

The result—for this program—is that the computed value of constit(s,0,2) will be the inside probability $\beta_{\mathsf{s}}(0,2)$ for a particular input sentence and grammar.[2] In practice one might wait until runtime to provide the description of the sentence (the rules for word and length) and perhaps even of the grammar (the rewrite axioms). In this case our transformations would typically be used only on the part of the program specified at compile time. But for simplicity, we suppose in this paper that the whole program is specified at compile time.

If the left-hand sides of two rules unify, then the rules must use the same aggregation operator, to guarantee that each item is aggregated in a consistent way. Each constit(...) item above is aggregated with +=.

4.2.3 Semantics of Weighted Logic Programs

In an unweighted logic program, the semantics is the set of provable items. For *weighted* logic programs, the semantics is a partial function that maps each provable item r to a value $[\![r]\!]$. All items in our example take values in $\mathbb{R}$. However, one could use values of any type or types.

The domain of the $[\![\cdot]\!]$ function is the set of items for which there exist finite proofs under the *unweighted* version of the program. We extend $[\![\cdot]\!]$ in the obvious way to expressions on provable items: for example, $[\![x * y]\!] \stackrel{\text{def}}{=} [\![x]\!] * [\![y]\!]$.

For each provable ground item r, let $\mathcal{P}(r)$ be the non-empty multiset of all ground expressions E on provable items such that $r \oplus_r\!= E$ instantiates some rule of $\mathcal{P}$. Here $\oplus_r\!=$ denotes the single aggregation operator shared by all those rules.

We now interpret the weighted rules as a set of simultaneous equations that constrain the $[\![\cdot]\!]$ function. If $\oplus_r\!=$ is +=, then we require that

$$[\![r]\!] = \sum_{E \in \mathcal{P}(r)} [\![E]\!]$$

(putting $[\![r]\!] = \infty$ if the sum diverges). More generally, we require that

$$[\![r]\!] = [\![E_1]\!] \oplus_r [\![E_2]\!] \oplus_r \ldots$$

where $\mathcal{P}(r) = \{E_1, E_2, \ldots\}$. For this to be well-defined, $\oplus_r$ must be associative and commutative. If $\oplus_r =$ is the special operator =, as in the final rules of our example, then we set $[\![r]\!] = [\![E_1]\!]$ if $\mathcal{P}(r)$ is a singleton set $\{E_1\}$, and generate an error otherwise.

In the terminology of the logic programming community, this definition is equivalent to saying that the valuation function $[\![\cdot]\!]$ is a fixed point of the monotone consequence operator.[3]

[2] However, unlike probabilistic programming languages (Zhou and Sato, 2003), we do not enforce that values be reals in [0, 1] or have probabilistic interpretations.

[3] Such a fixed point need not be unique, and there is a rich line of research into attempting

Example. In the example of section 4.2.2, this means that for any particular X, I, K for which constit(X,I,K) is provable, $[\![\text{constit}(X, I, K)]\!]$ equals

$$\begin{array}{rl} & \sum_{J,Y,Z} [\![\text{rewrite}(X,Y,Z)]\!] * [\![\text{constit}(Y,I,J)]\!] * [\![\text{constit}(Z,J,K)]\!] \\ + & \sum_{W} [\![\text{rewrite}(X,W)]\!] * [\![\text{word}(W,I,J)]\!] \end{array}$$

where, for example, the first summation ranges over term triples J, Y, Z such that the summand has a value. We sum over J,Y,Z because they do not appear in the rule's head constit(X,I,J), which is being defined.

Notation. We will henceforth adopt a convention of underlining any variables that appear only in a rule's body, to more clearly indicate the range of the summation. We will also underline variables that appear only in the rule's head; these indicate that the rule is not a dynamic programming rule.

Discussion. Substituting max= for += throughout the program would find Viterbi probabilities (best derivation) rather than inside probabilities (sum over derivations). Similarly, we can obtain the unweighted recognizer of section 4.2.1 by writing expressions over boolean values:[4]

constit(X,I,K) |= rewrite(X,$\underline{\text{Y}}$,$\underline{\text{Z}}$) & constit($\underline{\text{Y}}$,I,$\underline{\text{J}}$) & constit($\underline{\text{Z}}$,$\underline{\text{J}}$,K).

In general, this framework subsumes the practically useful case of Goodman (1999), which requires all values to fall in a single semiring and all rules to use only the semiring operations.[5]

Definition. A program transformation $T : \mathcal{P} \to \mathcal{P}'$ is defined to be **semantics-preserving** if for every item r which is provable by $\mathcal{P}$, r is also provable by $\mathcal{P}'$ and

$$[\![r]\!]_{\mathcal{P}} = [\![r]\!]_{\mathcal{P}'}$$

4.2.4 Computing Semantics by Forward-Chaining

A basic strategy for computing the semantics is "forward chaining." The idea is to maintain current values for all proved items, and to propagate updates to these values, from the right-hand side of a rule to its left-hand side, until all the equations are satisfied. (This might not halt: even an unweighted dynamic program can encode an arbitrary Turing machine.)

---

to more precisely characterize the intuitive semantics of logic programs with negation or aggregation. The interested reader should refer to Fitting (2002), or to, for example, Van Gelder (1992) or Ross and Sagiv (1992) for a discussion of the semantics of aggregate logic programs. In practice, one may obtain some single fixpoint by running the forward-chaining algorithm of the section 4.2.4 below and hoping that it converges.

[4] Using | for "or" and & for "and." The aggregation operators |= and &= can be regarded as implementing existential and universal quantification.

[5] Dropping these requirements allows our framework to handle neural networks, game trees, and other interesting systems of equations. Note that Goodman's "side conditions" can be easily handled in our framework (see Eisner et al., 2005).

As already noted in section 4.2.1, Shieber et al. (1995) gave a forward chaining algorithm (elsewhere called "semi-naive bottom-up evaluation") for unweighted *dynamic* programs. Eisner et al. (2005) extended this to handle the semiring-weighted case. Goodman (1999) gave a mixed algorithm.

Dealing with our full class of weighted logic programs—not just semiring-weighted dynamic programs—is a substantial generalization. The algorithm must propagate arbitrary updates, derive values for non-ground items, and obtain the value of foo(3,3), if not explicitly derived, from (e.g.) the derived value of foo(X,X) or foo(X,3) in preference to the less specific foo(X,Y). Furthermore, certain aggregation operators, but not all, permit optimizations that are important for efficiency. We defer these algorithmic details to a separate paper.

4.3 Folding

Weighted dynamic programs are schemata that define systems of simultaneous equations. Such systems can often be rearranged without affecting their solutions. In the same way, weighted dynamic programs can be transformed to obtain new programs with better runtime.

For a first example, consider our previous rule from section 4.2.2,

```
constit(X,I,K) += rewrite(X,Y,Z) * constit(Y,I,J) * constit(Z,J,K).
```

If the grammar has N nonterminals, and the input is an n-word sentence or an n-state lattice, then the above rule can be instantiated in only $O(N^3 \cdot n^3)$ different ways. For this—and the other parsing programs we consider here—it turns out the runtime of forward chaining can be kept down to $O(1)$ time per instantiation.[6] Thus the runtime is $O(N^3 \cdot n^3)$.

However, the following pair of rules is equivalent:

```
temp(X,Y,Z,I,J)   = rewrite(X,Y,Z)   * constit(Y,I,J).
constit(X,I,K)    += temp(X,Y,Z,I,J) * constit(Z,J,K).
```

We have just performed a weighted version of the classical **folding** transformation for logic programs (Tamaki and Sato, 1984). The original body expression would be explicitly parenthesized as (rewrite(X,Y,Z) * constit(Y,I,J)) * constit(Z,J,K); we have simply introduced a "temporary item" (like a temporary variable in a traditional language) to hold the result of the parenthesized subexpression, then "folded" that temporary item into the computation

[6] Assuming that the grammar is acyclic (in that it has no unary rule cycles) and so is the input lattice. Even without such assumptions, a meta-theorem of McAllester (1999) allows one to derive asymptotic runtimes of appropriately-indexed forward chaining from the number of instantiations. However, that meta-theorem applies only to unweighted dynamic programs. Similar results in the weighted case require acyclicity. Then one can use the two-phase method of Goodman (1999), which begins with a run of McAllester's method on an unweighted version of the program.

of constit. The temporary item mentions all the capitalized variables in the expression.

Distributivity. A more important use appears when we combine folding with the distributive law. In the example above, the second rule's body sums over the (underlined) free variables, J, Y, and Z. However, Y appears only in the temp item. We could therefore have summed over values of Y *before* multiplying by constit(Z,J,K), obtaining the following transformed program instead:

```
temp2(X,Z,I,J) += rewrite(X,Y,Z)  * constit(Y,I,J).
constit(X,I,K)   += temp2(X,Z,I,J) * constit(Z,J,K).
```

This version of the transformation is permitted only because + distributes over *.[7] By "forgetting" Y as soon as possible, we have reduced the runtime of CKY from $O(N^3 \cdot n^3)$ to $O(N^3 \cdot n^2 + N^2 \cdot n^3)$.

Using the distributive law to improve runtime is a well-known technique. Aji and McEliece (2000) present an algorithm inspired by the junction-tree algorithm for probabilistic inference in graphical models which they call the "generalized distributive law," which is equivalent to repeated application of the folding transformation, and which they demonstrate to be useful on a broad class of weighted logic programs.

A categorial grammar view of folding. From a parsing viewpoint, notice that the item temp2(X,Z,I,J) can be regarded as a categorial grammar constituent: an incomplete X missing a subconstituent Z at its right (i.e., an X/Z) that spans the substring from I to J. This leads us to an interesting and apparently novel way to write the transformed program:

```
constit(X,I,K)/constit(Z,J,K) += rewrite(X,Y,Z) * constit(Y,I,J).
constit(X,I,K) += constit(X,I,K)/constit(Z,J,K)  * constit(Z,J,K).
```

Here A/B is syntactic sugar for slash(A,B). That is, / is used as an infix functor and does not denote division, However, it is meant to *suggest* division: as the second rule shows, A/B is an item which, if multiplied by B, yields a summand of A. In effect, the first rule above is derived from the original rule at the start of this section by dividing both sides by constit(Z,J,K). The second rule multiplies the missing factor constit(Z,J,K) back in, now that the first rule has summed over Y.

Notice that K appears free (and hence underlined) in the head of the first rule. The only slashed items that are actually *provable* in this program are non-ground terms such as constit(s,0,K)/constit(n,1,K). That is, they have the form constit(X,I,K)/constit(Z,J,K) where X,I,J are ground variables but K remains free. The equality of the two K arguments (by internal unification) indicates that the missing Z is always at the *right* of the X, while their freeness means

[7] Since all semirings enforce a similar distributive property, the trick can be applied equally well to Viterbi parsing and unweighted recognition (section 4.2.3).

that the right edge of the full X and missing Z are still unknown (and will remain unknown until the second rule fills in a particular Z). Thus, the first rule performs a computation once for *all* possible K—the source of folding's efficiency. Our earlier program with temp2 could have been obtained by a further automatic transformation that replaced all constit(X,I,K)/constit(Z,J,K) having free K with the more compactly stored temp2(X,Z,I,J).

We emphasize that although our slashed items are inspired by categorial grammars, they can be used to describe folding in *any* weighted logic program. Section 4.5 will further exploit the analogy to obtain a novel "speculation" transformation.

Further applications. The folding transformation unifies various ideas that have been disparate in the literature. Eisner and Satta (1999) speed up parsing with bilexical context-free grammars from $O(n^5)$ to $O(n^4)$, using precisely the above trick (see section 4.4 below). Huang et al. (2005) employ the same "hook trick" to improve the complexity of syntax-based MT with an n-gram language model.

Another parsing application is the common "dotted rule" trick (Earley, 1970). If one's CFG contains ternary rules X → Y1 Y2 Y3, the naive CKY-like algorithm takes $O(N^4 \cdot n^4)$ time:

```
constit(X,I,L) += ((rewrite(X,Y1,Y2,Y3) * constit(Y1,I,J))
                    * constit(Y2,J,K)) * constit(Y3,K,L).
```

Fortunately, folding allows one to sum first over Y1 before summing separately over Y2 and J, and then over Y3 and K:

```
temp(X,Y2,Y3,I,J) += rewrite(X,Y1,Y2,Y3) * constit(Y1,I,J).
temp2(X,Y3,I,K)   += temp(X,Y2,Y3,I,J)   * constit(Y2,J,K).
constit(X,I,L)    += temp2(X,Y3,I,K)     * constit(Y3,K,L).
```

This restores $O(n^3)$ runtime (more precisely, $O(N^4 \cdot n^2 + N^3 \cdot n^3 + N^2 \cdot n^3)$)[8] by reducing the number of nested loops. Even if we had declined to sum over Y1 and Y2 in the first two rules, then the summation over J would already have obtained $O(n^3)$ runtime, in effect by binarizing the ternary rule. For example, temp2(X,Y1,Y2,Y3,I,K) would have corresponded to a partial constituent matching the *dotted* rule X → Y1 Y2 . Y3. The additional summations over Y1 and Y2 result in a more efficient dotted rule that "forgets" the names of the nonterminals matched so far, X → ? ? . Y3. This takes further advantage of distributivity by aggregating dotted-rule items (with +=) that will behave the same in subsequent computation.

The variable elimination algorithm for undirected graphical models can be viewed as repeated folding. An undirected graphical model expresses a joint

[8]For a dense grammar, which may have up to N^4 ternary rules. Tighter bounds on grammar size would yield tighter bounds on runtime.

probability distribution over P,Q by marginalizing (summing) over a product of clique potentials:

```
marginal(P,Q) += p1(...) * p2(...) * ··· * pn(...).
```

where a function such as p5(Q,X,Y) represents a clique potential over graph nodes corresponding to the random variables Q,X,Y. Assume without loss of generality that variable X appears as an argument only to $p_{k+1}, p_{k+2}, \ldots, p_n$. We may *eliminate* variable X by transforming to

```
temp(...)       += p_{k+1}(..., X, ...) * ··· * p_n(..., X, ...).
marginal(P,Q)   += p_1(...)             * ··· * p_k(...)          * temp(...).
```

The first line no longer mentions X because the second line sums over it. The variable elimination algorithm applies this procedure repeatedly to the last line to eliminate the remaining variables.[9]

Common subexpression elimination. Folding can also be used multiple times to eliminate common subexpressions. Consider the following code for *bilexical* CKY parsing:

```
constit(X:H,I,K) += rewrite(X:H,Y:H,Z:H2)
                    * constit(Y:H,I,J) * constit(Z:H2,J,K).
constit(X:H,I,K) += rewrite(X:H,Y:H2,Z:H)
                    * constit(Y:H2,I,J) * constit(Z:H,J,K).
```

Here X:H is syntactic sugar for ntlex(X,H), meaning a nonterminal X lexicalized at head word H. The program effectively has two types of rewrite rule, which pass the head word to the left or right child, respectively.

We could fold together the last two factors of the first rule to obtain

```
temp(Y:H,Z:H2,I,K) += constit(Y:H,I,J) * constit(Z:H2,J,K).
constit(X:H,I,K)   += rewrite(X:H,Y:H,Z:H2) * temp(Y:H,Z:H2,I,K).
constit(X:H,I,K)   += rewrite(X:H,Y:H2,Z:H)
                      * constit(Y:H2,I,J) * constit(Z:H,J,K).
```

We can *reuse* this definition of the temp rule to fold together the last two factors of line 3—which is the same subexpression, modulo variable renaming. (Below, for clarity, we explicitly and harmlessly swap the names of H2 and H inthe temp rule.)

```
temp(Y:H2,Z:H,I,K) += constit(Y:H2,I,J) * constit(Z:H,J,K).
constit(X:H,I,K)   += rewrite(X:H,Y:H,Z:H2) * temp(Y:H,Z:H2,I,K).
constit(X:H,I,K)   += rewrite(X:H,Y:H2,Z:H) * temp(Y:H2,Z:H,I,K).
```

Using the same temp rule (modulo variable renaming) in both folding transformations, rather than introducing a new temporary item for each fold, gives us a constant-factor improvement in time and space.

[9] Determining the optimal elimination order is NP-complete.

Definition of folding. Our definition allows an additional use of the distributive law. The original program may define the value of item r by aggregating values not only over free variables in the body of one rule, but also across n rules. Thus, when defining the temp item s, we also allow it to aggregate across n rules. In ordinary mathematical notation, we are performing a generalized version of the following substitution:

Before		**After**
$r = \sum_i (E_i * F)$	$\Rightarrow$	$r = s * F$
$s = \sum_i E_i$	$\Rightarrow$	$s = \sum_i E_i$

given the distributive property $\sum_i (E_i * F) = (\sum_i E_i) * F$. The common context in the original rules is the function "multiply by expression F," so the temp item s plays the role of r/F. We will generalize by allowing this common context to be an arbitrary function F.

We require that the rules defining the temp item, $s = \sum_i E_i$, be in the program *already* before folding occurs. If necessary, their presence may be arranged by a trivial **definition introduction** transformation that adds $r/F = \sum_i E_i$. (Explicitly using the slashed item r/F for s will ensure that the variable occurrence requirement below is met.) We claim without proof that all transformations in this paper are semantics-preserving in the sense of section 4.2.3.

Below and throughout the paper, we use the notation $F[X]$ to denote the *literal* substitution of expression X for all instances of μ in an expression F over items, even if X contains variables that appear in F or elsewhere in the rule containing $F[X]$. We assume that μ is a distinguished item name, of the same value type as X, and does not appear elsewhere.

Algorithm 4.3.1 (Folding transformation)

Given n distinct rules $R_1, \ldots, R_n$ in $\mathcal{P}$, where each R_i has the form $r \oplus= F[E_i]$. Given also a term s that unifies with the heads of exactly n rules in the program, all of which are distinct from the R_i, and which respectively take the form $s \odot= E_i$ after this unification.

Then the folding transformation deletes the rules $R_1, \ldots, R_n$, replacing them with a new rule $r \oplus= F[s]$, provided that

- *Any variable that occurs in any of the E_i which also occurs in either F or r must also occur in s.*[10]
- *Either $\oplus=$ or $\odot=$ is simply $=$,*[11] *or else the distributive property $[\![F[x \odot y]]\!] = [\![F[x]]\!] \oplus [\![F[y]]\!]$ holds for all assignments of terms to variables and all valuation functions $[\![\cdot]\!]$.*[12]

As a tricky example, one can replace r += p(I,J) * log(q(J,K)) with r += p(I,J) * log(s(J)) in the presence of s(J) *= q(J,K). Here E_1 is q(J,K), and F[x] is p(I,J) * log(x).

4.4 Unfolding

In general, a folding transformation leaves the asymptotic runtime alone, or may improve it when combined with the distributive law.[13] Thus, the inverse of the folding transformation, called **unfolding**, makes the asymptotic time complexity the same or worse. However, unfolding may be advantageous as a precursor to some other transformation that improves runtime. It also saves space. Sometimes we can improve both time and space complexity by unfolding and then transforming the program further.

For example, recall the bilexical CKY parser given near the end of section 4.3. The first rule originally shown there has runtime $O(N^3 \cdot n^5)$, since there are N possibilities for each of X,Y,Z and n possibilities for each of I,J,K,H,H2. Suppose that instead of that slow rule, the original programmer had written the following folded version:

```
temp3(X:H,Z:H2,I,J) += rewrite(X:H,Y:H,Z:H2) * constit(Y:H,I,J).
constit(X:H,I,K)    += temp3(X:H,Z:H2,I,J)   * constit(Z:H2,J,K).
```

This partial program has asymptotic runtime $O\left(N^3 \cdot n^4 + N^2 \cdot n^5\right)$ and needs $O\left(N^2 \cdot n^4\right)$ space to store the items (rule heads) it derives.

By unfolding the temp3 item—that is, substituting its definition in place each time it is used, which uses unification and relies on distributivity—and then trimming away its now-unneeded definition, we recover the first rule of the original program:

```
constit(X:H,I,K) += rewrite(X:H,Y:H,Z:H2)
                    * constit(Y:H,I,J) * constit(Z:H2,J,K).
```

This worsens the time complexity to $O\left(N^3 \cdot n^5\right)$, but by eliminating storage of the temp items, it improves the space complexity to $O\left(N \cdot n^3\right)$. The payoff is that now we can refold this rule differently—either as in section 4.3, or alternatively as follows (Eisner and Satta, 1999, which misses the chance to eliminate common subexpressions):

[10] This ensures that s does not sum over any variables that must remain visible in the revised r rule.

[11] For instance, in the very first example of section 4.3, the temp item was defined using = and therefore performed no summation. No distributivity was needed.

[12] That is, all valuation functions over the space of items, including dummy items x and y, when extended over expressions in the usual way.

[13] It may either help or hurt the *actual* runtime, and it certainly increases the space needed to store items' values.

```
temp4(X:H,Y:H,J,K) += rewrite(X:H,Y:H,Z:H2) * constit(Z:H2,J,K).
constit(X:H,I,K)   += temp4(X:H,Y:H,J,K)    * constit(Y:H,I,J).
```

Either way, the time complexity is now $O\left(N^3 \cdot n^4 + N^2 \cdot n^4\right)$—better than the original programmer's version—while the space complexity has increased only back to the original programmer's $O\left(N^2 \cdot n^4\right)$.

Unfolding resembles inlining of a subroutine call. Section 4.5 will show how it can thus be used for program specialization—improving efficiency by a constant factor and also enabling further transformations that improve asymptotic efficiency.

4.5 Speculation

We now generalize folding to handle recursive rules. This **speculation** transformation, which is novel as far as we know, is reminiscent of gap-passing in categorial grammar. It has many uses; we limit ourselves to two examples.

Split head-automaton grammars. We consider a restricted kind of bilexical CFG in which a head word combines with all of its right children before any of its left children (Eisner and Satta, 1999). The "inside algorithm" below[14] builds up rconstit items by starting with a word and successively adding 0 or more child constituents to the right, then builds up constit items by adding 0 or more child constituents to the left of this.

```
rconstit(X:H,I,K) += word(H,I,K).                  % 0 right children so far
rconstit(X:H,I,K) += rewrite(X:H,Y:H,Z:H2)         % add right child
                     * rconstit(Y:H,I,J) * constit(Z:H2,J,K).
constit(X:H,I,K)  += rconstit(X:H,I,K).            % 0 left children so far
constit(X:H,I,K)  += rewrite(X:H,Y:H2,Z:H)         % add left child
                     * constit(Y:H2,I,J) * constit(Z:H,J,K).
goal += constit(s:H,0,N) * length(N).
```

This algorithm has runtime $O(N^3 \cdot n^5)$ (dominated by line 4). We now exploit the conditional independence of left children from right children. Instead of building up a constit from a particular, existing rconstit (line 3) and then adding left children (line 4), we transform the program so it builds up the constit item *speculatively*, waiting until the end to fill in each of the various rconstit items that could have spawned it. Replace lines 3–4 with

```
lconstit(X0:H0,X0,J0,J0) += 1   needed_only_if  rconstit(X0:H0,J0,K0).
lconstit(X:H0,X0,I,J0)   += rewrite(X:H0,Y:H2,Z:H)
                            * constit(Y:H2,I,J) * lconstit(Z:H0,X0,J,J0).
                            needed_only_if  rconstit(X0:H0,J0,K0).
constit(X:H0,I,K0)       += lconstit(X:H0,X0,I,J0) * rconstit(X0:H0,J0,K0).
```

[14]For simplicity, this code ignores the cost of starting, "flipping," or stopping in different nonterminal states.

The new temp item lconstit(X:H0,X0,I,J0) represents the *left* half of a constituent. We can regard it in the categorial terms of section 4.3: as the last line illustrates, it is just a more compact notation for a constit missing its rconstit right half—namely constit(X:H0,I,K0)/rconstit(X0:H0,J0,K0), where K0 is always a free variable, so that lconstit need not specify any particular value for K0.

The first lconstit rule introduces an empty left half, equivalent to constit(X0:H0,J0,K0)/rconstit(X0:H0,J0,K0). This is extended with its left children by recursing through the second lconstit rule, allowing X and I to diverge from X0 and J0 respectively. Finally, the last rule finally fills in the missing right half rconstit.

The special filter clauses needed_only_if rconstit(X0:H0,J0,K0) are added solely for efficiency. They say that it is not necessary to build "useless" left halves purely speculatively, but only when there is some right half for them to combine with. Their semantics are sketched below.

In this case, the filter clause on the second rule manages to ensure that in any lconstit(X:H, X0,I,J0) that we need to build, J0 will be the start position of the head word H. (Such a constraint is already true for rconstits; an empty lconstit inherits it from the rconstit filter, and passes it along to successively wider lconstit.) Since the temp item records only this redundant position and not K (the right boundary of the unknown rconstit), runtime falls from $O\left(n^5\right)$ to $O\left(n^4\right)$.

As a bonus, we can now obtain the $O\left(n^3\right)$ algorithm of Eisner and Satta (1999). Simply unfold the instances of constit in the rconstit and temp rules (i.e., replacing them with lconstit * rconstit per our new definition). Then refold those rules differently.[15]

Filter clauses. Our approach to filtering is novel. Our needed_only_if clauses may be regarded as "relaxed" versions of side conditions (Goodman, 1999). In the denotational semantics (section 4.2.3), they relax the restrictions on the $[\![\cdot]\!]$ function, allowing more possible semantics (all of which, however, preserve the semantics of the original program).

Specifically, when constructing $\mathcal{P}(r)$ to determine whether a ground item r is provable and what its value is, we may *optionally* omit an instantiated rule $r \oplus_r= E$ if it has a filter clause needed_only_if C such that no consistent instantiation of C has been proved. (The "consistent" instantiations are those where variables of C that are shared with r or E are instantiated accordingly. Other variables, such as K0 in the example above, may have any instantiation.)

How does this help operationally, in the forward chaining algorithm?

[15] In each case, use a rewrite to combine a rconstit with an lconstit to its right (first folding the rewrite with whichever one does not contribute the head word).

When a rule triggers an update to a ground *or* non-ground item, but carries a (partly instantiated) filter clause that does not unify with any proved item, then the update has infinitely low priority and need not be placed on the forward-chaining agenda. The update must still be carried out if the filter clause is proved later.[16]

In the example above, forward chaining on the first lconstit rule produces an "zero-width" lconstit(X0:H0,X0,J0,J0) in which all variables are free.[17] This lconstit can be used anywhere; in particular, it can combine with any rconstit, so the filter clause says it is needed as soon as *any* rconstit has been proved. The real filtering power comes when the second rule tries to build further from the zero-width lconstit using the second rule. Then X0, H0, and J0 indirectly become bound to values in the rewrite and constit items of that rule (because of the internal unification in the zero-width lconstit(X0:H0,X0,J0,J0)). Thus, the filter clause is now better instantiated, e.g., needed_only_if rconstit(vp:"flies",1,K0). Only if such an rconstit has been derived (for some K0) are we required to consider updating the clause head, e.g., lconstit(s:"flies",vp,0,1).

Unary rule closure. Before formalizing speculation, we informally show another instructive application: precomputing unary rule closure in a CFG. We start with a version of the inside algorithm that allows nonterminal unary rules:

program fragment $\mathcal{P}_0$:
constit(X,I,K) += rewrite(X,<u>W</u>) * word(<u>W</u>,I,K).
constit(X,I,K) += rewrite(X,<u>Y</u>) * constit(<u>Y</u>,I,K).
constit(X,I,K) += rewrite(X,<u>Y</u>,<u>Z</u>) * constit(<u>Y</u>,I,<u>J</u>) * constit(<u>Z</u>,<u>J</u>,K).

Suppose that the grammar rules include, among others,

program fragment $\mathcal{P}_0$: (continued)
rewrite(np1,np3) = 0.1.
rewrite(np3,np2) = 0.2.
rewrite(np2,np3) = 0.3.
rewrite(np3,det,n)= 0.4. . . .

We can *unfold* the grammar into the program to get rules such as

program fragment $\mathcal{P}_1$:
constit(np1,I,K) += 0.1 * constit(np3, I, K).
constit(np3,I,K) += 0.2 * constit(np2, I, K).
constit(np2,I,K) += 0.3 * constit(np3, I, K).

---

[16] Sometimes a filter is true, causing the update, but later becomes false. For instance, rconstit(vp:"flies",1,K0) may no longer be provable after sentence-specific word(. . .) axioms are retracted. Because the update is now optional, the algorithm is not required to retract the update (at least not on that basis), although it is free to do so in order to reclaim memory. This optionality is useful in some of our examples below: entries will be filled into the unary-rule-closure and left-corner tables only as needed, but need not be retracted after each sentence and then rederived.

[17] As well as adding 1 to any other items that specialize and override this one.

```
constit(np3,I,K) += 0.4 * constit(det, I, J) * constit(n, J, K). . . .
```

This amounts to program specialization. If we have unfolded (at least) the unary rewrite rules into the program, we can now apply speculation to eliminate them "offline":

program fragment $\mathcal{P}_2$*:*

```
temp(X0,X0)      += 1   needed_only_if  constit(X0,I0,K0).
temp(np1,X0)     += 0.1 * temp(np3,X0).
temp(np3,X0)     += 0.2 * temp(np2,X0).
temp(np2,X0)     += 0.3 * temp(np3,X0).
constit(X,I0,K0) += temp(X,X0) * other(constit(X0,I0,K0)).
other(constit(np3,I,K)) += 0.4 * constit(det,I,J) * constit(n,J,K). . . .
```

For any nonterminals X and Y, our temporary item temp(X,X0) is just compact notation for constit(X,I0,K0)/constit(X0,I0,K0): the inside probability of deriving a constit(X,I0,K0) by a sequence of 0 or more unary rules from a constit(X0,I0,K0) that covers the same span I0–K0. In other words, it is the total probability of all (possibly empty) unary-rewrite chains X $\rightarrow^*$ X0.

The final two rules recover unslashed constit items. other(constit(X,I,K)) is any constit(X0,I,K) whose derivation does *not* begin with a unary rule. The next-to-last rule builds this into constit(X,I,J) through a sequence of 0 or more unary rules.

Crucially, the temp(X,X0) items have values that are *independent of* I and K. So they need not be computed separately for every span in every sentence. For each nonterminal X0, all temp(X,X0) values will be computed once and for all (the very first time a constit(X0,I,K) constituent is built) by iterating the first three rules below to convergence. These values will then remain static while the grammar does, even if the sentence changes (see footnote 16).

Definition of speculation. In general, the value of a slashed item is a *function*, just like the semantics of a slashed constituent in categorial grammar. Also as in categorial grammar, gaps are introduced with the identity function, passed with function composition, and eliminated with function application. Fortunately, in commutative semiring-weighted programs like the ones above, all functions have the form "multiply by x" for some weight x. We can represent such a function simply as x, using semiring 1 for the identity function, semiring multiplication for both composition and application, and semiring addition for pointwise addition.

Algorithm 4.5.1 (Speculation transformation)

Let a be an item to slash out, where any variables in a do not occur elsewhere in $\mathcal{P}$. Let slash *and* other *be functors that do not already appear in $\mathcal{P}$. Let $R_1, \ldots, R_n$ be distinct rules in $\mathcal{P}$, where each R_i is $r_i \oplus_i = F_i[t_i]$, and*
- *For $i \leq k$, t_i does not unify with a.*
- *For $i > k$, t_i unifies with a; more strongly, it matches a non-empty subset of the ground terms that a does.*[18]
- *Certain conditions on distributivity (satisfied by semiring programs).*

Then the speculation transformation constructs the following new program, in which the values of slash *items are functions, $\oplus_i$ is extended to sum functions pointwise, $\circ$ denotes function composition, and $F[x]$ denotes function application.*
- *slash(a,a)* $\oplus_{\overline{a}}=$ $(\lambda x.\ x)$ *needed_only_if* a.
- $(\forall 1 \leq i \leq n)$ *slash(r_i, a)* $\oplus_i = F_i \circ$ *slash(t_i, a) needed_only_if* a.
- $(\forall 1 \leq i \leq k)$ *other(r_i)* $\oplus_i = F_i$*[other(t_i)].*
- $(\forall$ *rules p* $\oplus=$ *q not among the* $R_i)$ *other(p)* $\oplus=$ *q.*[19]
- *X* $\oplus_X=$ *other(X) unless X is an instance of a.*
- *X* $\oplus_X=$ *(slash(X,a))[other(a)].*[20]

Intuitively, other(X) accumulates ways of building X other than instantiations of $F_{i_1}[F_{i_2}[\cdots F_{i_j}[a]]]$ for $j > 0$. slash(X,a) aggregates all instantiations of the function $\lambda x.F_{i_1}[F_{i_2}[\cdots F_{i_j}[x]]]$ for $j \geq 0$. This pointwise sum of functions is only applied to other(...) items, to prevent double-counting (analogous to spurious ambiguity in a categorial grammar).

To apply this formal transformation in the unary-rule elimination example, take a=constit(X0,I0,K0), and the R_i to be the "unary" constit rules, where each t_i is the last item in the body of R_i. Here $k = 0$. The resulting slashed items have the form slash(constit(X,I,K), constit(X0,I0,K0)), but the rules would only derive instances where I=I0 and K=K0. All such rules are filtered by needed_only_if constit(X0,I0,K0).[21]

To apply the transformation in the split head-automaton example, take a=constit(X0:H0,J0,K0), the R_i to be the two rules defining constit, each t_i to be the last item in the body of R_i, and $k = 1$.[22]

[18]By adding side conditions, any rule can be split into an $i \leq k$ rule, an $i > k$ rule, and a rule not among the R_i.

[19]Typically, many of the other(...) items can be unfolded and then their defining rules removed. This is why few or none remained in the informal examples above.

[20]In the final two rules, X ranges over the entire universe of terms. Recall that $\oplus_X$ is the aggregation operator for X. One could construct separate rules for items aggregated with different operators.

[21]In this example, the efficiency filters are redundant on rules after the first. Runtime analysis or (perhaps) static analysis would show that they have no actual filtering effect, allowing us to eliminate them.

[22]In this program, *all* constits are built from rconstits using R_1 and R_2, so other(constit(...)) has no derivations. Concretely, the single rule that the transformation generates to define

4.6 Converting bottom-up to top-down

4.6.1 Magic Templates

Finally, we give an important transformation that explains and generalizes the way that speculation introduced needed_only_if filters.

The bottom-up "forward-chaining" execution strategy mentioned in section 4.2.4 will compute the values for all provable items. Many of these items may, however, be irrelevant in the sense that they do not contribute directly or indirectly to the value of goal. (In parsing, they are legal constituents that do not lead to a complete parse.) We can avoid generation of these irrelevant items by employing the magic templates transformation (Ramakrishnan, 1991), which prevents an item from being built unless it will help lead to a "desired" item.

We need the value of a theorem foo if it occurs in in the body of a rule where (1) we need the value of the rule's head and (2) we have already derived the items preceding foo in the rule's body.[23] For example, in the CKY parsing rule

constit(X,I,K) += rewrite(X,$\underline{\text{Y}}$,$\underline{\text{Z}}$) * constit($\underline{\text{Y}}$,I,$\underline{\text{J}}$) * constit($\underline{\text{Z}}$,$\underline{\text{J}}$,K).

we need constit(Y,I,J) (for a particular Y,I,J) if we need constit(X,I,K) (for some X,K) and we already know that rewrite(X,Y,Z) is provable (for some Z), which we denote ?rewrite(X,Y,Z).[24] Hence

magic(constit(Y,I,$\underline{\text{J}}$)) | = magic(constit($\underline{\text{X}}$,I,$\underline{\text{K}}$)) & ?rewrite($\underline{\text{X}}$,Y,$\underline{\text{Z}}$).

For example, the above rule may derive magic(constit(vp,1,J)) as true. That means it is worthwhile to look for vp objects *starting* at position 1. The *ending* position J is unspecified—a free variable. Ramakrishnan (1991)'s original presentation drops such superfluous variables to obtain a dynamic programming version:

magic_constit(Y,I)) | = magic(constit($\underline{\text{X}}$,I,$\underline{\text{K}}$)) & ?rewrite($\underline{\text{X}}$,Y,$\underline{\text{Z}}$).

Ramakrishnan's move is not necessary for the present section, but it improves efficiency, and will simplify section 4.6.2.

Here are all the magic rules for CKY parsing (section 4.2.2):

```
magic_goal          | = true.
magic_constit(s,0)  | = magic_goal.
```

other(constit(...)) depends on other(constit(...)) itself, so it can never be derived from the axioms (and may be trimmed away as useless).

[23] This left-to-right order within a rule is traditional, but any order would do.

[24] It would not be appropriate to write needed_only_if rewrite(X,Y,Z). The rewrite item is part of the definition of whether the magic item should be true or false—not simply a condition on whether a more lenient definition of magic (which would serve as a less effective filter) is worth deriving. As a concrete consequence, using needed_only_if rewrite(X,Y,Z) below would derive a magic item in which Y remained a free variable, a lenient definition that would license the main program to derive many useless constit items.

```
magic_constit(Y,I)    | = magic_constit(X,I,K)) & ?rewrite(X,Y,Z).
magic_constit(Z,J)    | = magic_constit(X,I)
                            & ?rewrite(X,Y,Z) & ?constit(Y,I,J).
```

Then, we modify the rules of the original program, adding magic_foo as a filter on the derivation of foo:

```
constit(X,I,K) + = rewrite(X,W) * word(W,I,K)
                   needed_only_if magic_constit(X,I).
constit(X,I,K) + = rewrite(X,Y,Z) * constit(Y,I,J) * constit(Z,J,K)
                   needed_only_if magic_constit(X,I).
goal           + = constit(s,0,N) * length(N) needed_only_if magic_goal.
```

This transformed program uses forward chaining to simulate backward chaining (though perhaps a breadth-first version of backward chaining). Since we ultimately want the value of goal (or derivations of goal), we set magic_goal=true. That causes us to derive magic_constit facts at the start of the sentence, which license the building of actual constit items with values, which let us derive magic_constit facts later in the sentence, and so on. Remarkably, as previously noticed by Minnen (1996), the operation of this transformed program is the same as Earley's algorithm (Earley, 1970): constituents are predicted top-down, and built bottom-up only if they have a "customer" to the immediate left.

Shieber et al. (1995), specifying CKY and Earley's algorithm, remark that "proofs of soundness and completeness [for the Earley's case] are somewhat more complex ... and are directly related to the corresponding proofs for Earley's original algorithm." In our perspective, the correctness of Earley's emerges directly from the correctness of CKY and the semantics-preserving nature of the magic templates transformation.

Another application is "on-the-fly" intersection of weighted finite-state automata, which recalls the left-to-right nature of Earley's algorithm. Intersection of arcs $Q \xrightarrow{X} R$ in machines M_1 and M_2, bearing the same symbol X, is accomplished by multiplying their weights:

```
arc(M1:M2,Q1:Q2,R1:R2,X) += arc(M1,Q1,R1,W) * arc(M2,Q2,R2,X).
```

But this pairs all compatible arcs in all known machines (including the new machine M1:M2, leading to infinite regress). A magic templates transformation can restrict to arcs that actually need to be derived in the service of some larger goal (e.g., summing over selected paths from a specified paired start state Q1:Q2).

4.6.2 Second-order magic

Using magic templates to change to a top-down computation order will still allow some irrelevant items to be derived. Not all items we "need" to derive a value for goal, according to a top-down search from goal, will actually turn out to be provable bottom-up. This may lead to too much top-down exploration: Earley's algorithm may predict many categories such as vp at position 1 (i.e., derive magic(constit(vp,1,J))) when there is not even a possible verb at position 1.

We can therefore apply the magic templates transformation a second time, to the rules that defined the first-order magic items. This yields second-order magic items of the form magic_magic_foo, meaning "we need to realize that we need to build foo":

```
magic_magic_goal          | = magic_magic_constit(s,0).
magic_magic_constit(X,I)  | = magic_magic_constit(Y,I) & ?rewrite(X,Y,Z).
magic_magic_constit(X,I)  | = magic_magic_constit(Z,J)
                              & ?rewrite(X,Y,Z) & ?constit(Y,I,J).
```

They can be added as needed_only_if filter clauses that limit Earley's "predict" rules (i.e., the rules that derive the first-order magic items). As before, K remains free. Consider in particular the second rule above, which says that if Earley's can wisely predict Y at position I, it can also wisely predict X and (by recursion) any other nonterminal of which Y is a left corner. (Using speculation to abstract away from the sentence position I, we could build up a left corner table offline.)

The base case of this left-corner computation comes from enchanting one of the rules that *uses* rather than *defines* a first-order magic item,[25]

```
constit(X,I,K) + = rewrite(X,W) * word(W,I,K)
                   needed_only_if magic_constit(X,I).
```

to obtain

```
magic_magic_constit(X,I) | = ?rewrite(X,W) & ?word(W,I,K).
```

Thus, the second-order predicates will constrain top-down prediction at position I to predict only nonterminals that are left-corner compatible with the word W at I. In short, we have derived the left-corner filter on Earley's algorithm, by repeating the same transformation that derived Earley's algorithm in the first place!

4.7 Conclusions

We introduced a weighted logic programming formalism for describing a wide range of useful algorithms. After sketching its denotational and operational semantics, we outlined a number of fundamental techniques—program transformations—for rearranging a weighted logic program to make it more efficient.

[25]We do not show the enchantments of the other such rules, as they do not add any further power.

In addition to exploiting several known logic programming transformations, we described a weighted extension of folding and unfolding, and presented the speculation transformation, a substantial generalization of folding.

We showed that each technique was connected to ideas in both logic programming and in parsing, and had multiple uses in NLP algorithms. We recovered several known parsing optimizations by applying reusable transformations: for example, Earley's algorithm, the left-corner filter, parser specialization, offline unary rule cycle elimination, and the bilexical parsing techniques from (Eisner and Satta, 1999).

We noted throughout how program transformations could be simplified by allowing the resulting programs to derive non-ground items. One important tool was our proposed needed_only_if filter.

The paradigm and techniques presented here may be directly useful to algorithm designers as well as to those who are interested in formalisms for specifying and manipulating algorithms.

References

Aji, S. and R. McEliece. 2000. The generalized distributive law. *IEEE Transactions on Information Theory* 46(2):325–343.

Earley, J. 1970. An efficient context-free parsing algorithm. *Comm. ACM* 13(2):94–102.

Eisner, J., E. Goldlust, and N. A. Smith. 2005. Compiling comp ling: Weighted dynamic programming and the Dyna language. In *Proc of HLT/EMNLP*.

Eisner, J. and G. Satta. 1999. Efficient parsing for bilexical context-free grammars and head-automaton grammars. In *Proc. of ACL*, pages 457–464.

Fitting, M. 2002. Fixpoint semantics for logic programming a survey. *TCS* 278(1-2):25–51.

Goodman, J. 1999. Semiring parsing. *Computational Linguistics* 25(4):573–605.

Huang, L., H. Zhang, and D. Gildea. 2005. Machine translation as lexicalized parsing with hooks. In *Proc. of IWPT*, pages 65–73.

McAllester, D. 1999. On the complexity analysis of static analyses. In *Proc of 6th Internat. Static Analysis Symposium*.

Minnen, G. 1996. Magic for filter optimization in dynamic bottom-up processing. In *Proc 34th ACL*, pages 247–254.

Ramakrishnan, R. 1991. Magic templates: a spellbinding approach to logic programs. *J. Log. Prog.* 11(3-4):189–216.

Ross, K. A. and Y. Sagiv. 1992. Monotonic aggregation in deductive databases. In *PODS '92*, pages 114–126.

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

Tamaki, H. and T. Sato. 1984. Unfold/fold transformation of logic programs. In *Proc 2nd ICLP*, pages 127–138.

Van Gelder, A. 1992. The well-founded semantics of aggregation. In *PODS '92*, pages 127–138. New York, NY, USA: ACM Press. ISBN 0-89791-519-4.

Younger, D. H. 1967. Recognition and parsing of context-free languages in time n^3. *Info. and Control* 10(2):189–208.

Zhou, N.-F.. and T. Sato. 2003. Toward a high-performance system for symbolic and statistical modeling. In *Proc of IJCAI Workshop on Learning Stat. Models from Relational Data*.

5

On Theoretical and Practical Complexity of TAG Parsers

Carlos Gómez-Rodríguez, Miguel A. Alonso, Manuel Vilares

Abstract

We present a system allowing the automatic transformation of parsing schemata to efficient executable implementations of their corresponding algorithms. This system can be used to easily prototype, test and compare different parsing algorithms. In this work, it has been used to generate several different parsers for Context Free Grammars and Tree Adjoining Grammars. By comparing their performance on different sized, artificially generated grammars, we can measure their empirical computational complexity. This allows us to evaluate the overhead caused by using Tree Adjoining Grammars to parse context-free languages, and the influence of string and grammar size on Tree Adjoining Grammars parsing.

Keywords Parsing Schemata, Computational Complexity, Tree Adjoining Grammars, Context Free Grammars

5.1 Introduction

The process of parsing, by which we obtain the structure of a sentence as a result of the application of grammatical rules, is a highly relevant step in the automatic analysis of natural languages. In the last decades, various parsing algorithms have been developed to accomplish this task. Although all of these algorithms essentially share the common goal of generating a tree structure describing the input sentence by means of a grammar, the approaches used to attain this result vary greatly between algorithms, so that different parsing algorithms are best suited to different situations.

Parsing schemata, introduced in (Sikkel, 1997), provide a formal, simple and uniform way to describe, analyze and compare different parsing algo-

FG-2006.
Edited by Paola Monachesi, Gerald Penn,
Giorgio Satta and Shuly Wintner.

rithms. The notion of a parsing schema comes from considering parsing as a deduction process which generates intermediate results called *items*. An initial set of items is directly obtained from the input sentence, and the parsing process consists of the application of inference rules (called *deductive steps*) which produce new items from existing ones. Each item contains a piece of information about the sentence's structure, and a successful parsing process will produce at least one *final item* containing a full parse tree for the sentence or guaranteeing its existence.

Almost all known parsing algorithms may be described by a parsing schema (non-constructive parsers, such as those based on neural networks, are exceptions). This is done by identifying the kinds of items that are used by a given algorithm, defining a set of inference rules describing the legal ways of obtaining new items, and specifying the set of final items.

As an example, we introduce a CYK-based algorithm (Vijay-Shanker and Joshi 1985) for Tree Adjoining Grammars (TAG) (Joshi and Schabes 1997). Given a tree adjoining grammar $G = (V_T, V_N, S, I, A)$[1] and a sentence of length n which we denote by $a_1\ a_2\ \ldots\ a_n$[2], we denote by $P(G)$ the set of productions $\{N^\gamma \rightarrow N_1^\gamma N_2^\gamma \ldots N_r^\gamma\}$ such that N^γ is an inner node of a tree $\gamma \in (I \cup A)$, and $N_1^\gamma N_2^\gamma \ldots N_r^\gamma$ is the ordered sequence of direct children of N^γ.

The parsing schema for the TAG CYK-based algorithm is a function that maps such a grammar G to a deduction system whose domain is the set of items

$$\{[N^\gamma, i, j, p, q, adj]\}$$

verifying that N^γ is a tree node in an elementary tree $\gamma \in (I \cup A)$, i and j ($0 \leq i \leq j$) are string positions, p and q may be undefined or instantiated to positions $i \leq p \leq q \leq j$ (the latter only when $\gamma \in A$), and $adj \in \{true, false\}$ indicates whether an adjunction has been performed on node N^γ.

The positions i and j indicate that a substring $a_{i+1} \ldots a_j$ of the string is being recognized, and positions p and q denote the substring dominated by γ's foot node. The final item set would be

$$\{[R^\alpha, 0, n, -, -, adj] \mid \alpha \in I\}$$

for the presence of such an item would indicate that there exists a valid parse tree with yield $a_1\ a_2\ \ldots\ a_n$ and rooted at R^α, the root of an initial tree; and therefore there exists a complete parse tree for the sentence.

---

[1] Where V_T denotes the set of terminal symbols, V_N the set of nonterminal symbols, S the axiom, I the set of initial trees and A the set of auxiliary trees.

[2] From now on, we will follow the usual conventions by which nonterminal symbols are represented by uppercase letters ($A, B \ldots$), and terminals by lowercase letters ($a, b \ldots$). Greek letters (α, β...) will be used to represent trees, N^γ a node in the tree γ, and R^γ the root node of the tree γ.

A *deductive step* $\frac{\eta_1 \ldots \eta_m}{\xi}\ \Phi$ allows us to infer the item specified by its consequent ξ from those in its antecedents $\eta_1 \ldots \eta_m$. *Side conditions* (Φ) specify the valid values for the variables appearing in the antecedents and consequent, and may refer to grammar rules or specify other constraints that must be verified in order to infer the consequent. An example of one of the schema's deductive steps would be the following, where the operation $p \cup p'$ returns p if p is defined, and p' otherwise:

$$\text{CYK Binary: } \frac{\begin{array}{c}[O_1^\gamma, i, j', p, q, adj1] \\ [O_2^\gamma, j', j, p', q', adj2]\end{array}}{[M^\gamma, i, j, p \cup p', q \cup q', false]}\ M^\gamma \to O_1^\gamma O_2^\gamma \in P(G)$$

This deductive step represents the bottom-up parsing operation which joins two subtrees into one, and is analogous to one of the deductive steps of the CYK parser for Context-Free Grammars (Kasami 1965, Younger 1967). The full TAG CYK parsing schema has six deductive steps (or seven, if we work with TAGs supporting the substitution operation) and can be found at (Alonso et al., 1999). However, this sample deductive step is an example of how parsing schemata convey the fundamental semantics of parsing algorithms in simple, high-level descriptions. A parsing schema defines a set of possible intermediate results and allowed operations on them, but doesn't specify data structures for storing the results or an order for the operations to be executed.

5.2 Compilation of parsing schemata

Their simplicity and abstraction of low-level details makes parsing schemata very useful, allowing us to define parsers in a simple and straightforward way. Comparing parsers, or considering aspects such as their correction and completeness or their computational complexity, also becomes easier if we think in terms of schemata.

However, the problem with parsing schemata is that, although they are very useful when designing and comparing parsers with pencil and paper, they cannot be executed directly in a computer. In order to execute the parsers and analyze their results and performance they must be implemented in a programming language, making it necessary to abandon the high abstraction level and focus on the implementation details in order to obtain a functional and efficient implementation.

In order to bridge this gap between theory and practice, we have designed and implemented a compiler able to automatically transform parsing schemata into efficient Java implementations of their corresponding algorithms. The input to this system is a simple and declarative representation of a parsing schema, which is practically equal to the formal notation that we used previously. For example, this is the CYK deductive step we have seen as an example in a format readable by our compiler:

```
@step CYKBinary
[ Node1 , i , j' , p , q , adj1 ]
[ Node2 , j' , j , p' , q' , adj2 ]
----------------------------------------------------- Node3 -¿ Node1 Node2
[ Node3 , i , j , Union(p;p') , Union(q;q') , false ]
```

The parsing schemata compilation technique behind our system is based on the following fundamental ideas:

- Each deductive step is compiled to a Java class containing code to match and search for antecedent items and generate the corresponding conclusions from the consequent.
- The generated implementation will create an instance of this class for each possible set of values satisfying the side conditions that refer to production rules. For example, a distinct instance of the CYK Binary step will be created for each grammar rule of the form $M^\gamma \rightarrow O_1^\gamma O_2^\gamma \in P(G)$, as specified in the step's side condition.
- The step instances are coordinated by a deductive parsing engine, as the one described in (Shieber et al., 1995). This algorithm ensures a sound and complete deduction process, guaranteeing that all items that can be generated from the initial items will be obtained. It is a generic, schema-independent algorithm, so its implementation is the same for any parsing schema. The engine works with the set of all items that have been generated and an *agenda*, implemented as a queue, holding the items we have not yet tried to trigger new deductions with.
- In order to attain efficiency, an automatic analysis of the schema is performed in order to create indexes allowing fast access to items. Two kinds of index structures are generated: *existence indexes* are used by the parsing engine to check whether a given item exists in the item set, while *search indexes* are used to search for all items conforming to a given specification. As each different parsing schema needs to perform different searches for antecedent items, the index structures that we generate are schema-specific. Each deductive step is analyzed in order to keep track of which variables will be instantiated to a concrete value when a search must be performed. This information is known at schema compilation time and allows us to create indexes by the elements corresponding to instantiated variables. In this way, we guarantee constant-time access to items so that the computational complexity of our generated implementations is never above the theoretical complexity of the parsing algorithms.
- *Deductive step indexes* are also generated to provide efficient access to the set of deductive step instances which can be applicable to a given item.

Step instances that are known not to match the item are filtered out by these indexes, so less time is spent on unsuccessful item matching.

- Since parsing schemata have an open notation, for any mathematical object can potentially appear inside items, the system includes an extensibility mechanism which can be used to define new kinds of objects to use in schemata. The code generator can deal with these user-defined objects as long as some simple and well-defined guidelines are followed in their specification.

A more detailed description of this system, including a more thorough explanation of automatic index generation, can be found at (Gómez-Rodríguez et al., 2006b).

5.3 Parsing natural language CFG's

Although our main focus in this paper is on performance of TAG parsing algorithms, we will briefly outline the results of some experiments on Context-Free Grammars (CFG), described in further detail in (Gómez-Rodríguez et al., 2006b), in order to be able to contrast TAG and CFG parsing.

Our compilation technique was used to generate parsers for the CYK (Kasami 1965, Younger 1967), Earley (Earley 1970) and Left-Corner (Rosenkrantz and Lewis II 1970) algorithms for context-free grammars, and these parsers were tested on automatically-generated sentences from three different natural language grammars: Susanne (Sampson 1994), Alvey (Carroll 1993) and Deltra (Schoorl and Belder 1990). The runtimes for all the algorithms and grammars showed an empirical computational complexity far below the theoretical worst-case bound of $O(n^3)$, where n denotes the length of the input string. In the case of the Susanne grammar, the measurements were close to being linear with string size. In the other grammars, the runtimes grew faster, approximately $O(n^2)$, still far below the cubic worst-case bound.

Another interesting result was that the CYK algorithm performed better than the Earley-type algorithms in all cases, despite being generally considered slower. The reason is that these considerations are based on time complexity relative to string length, and do not take into account time complexity relative to grammar size, which is $O(|P|)$ for CYK and $O(|P|)^2$ for the Earley-type algorithms. This factor is not very important when working with small grammars, such as the ones used for programming languages, but it becomes fundamental when we work with natural language grammars, where we use thousands of rules (more than 17,000 in the case of Susanne) to parse relatively small sentences. When comparing the results from the three context-free grammars, we observed that the performance gap between CYK and Earley was bigger when working with larger grammars.

5.4 Parsing artificial TAG's

In this section, we make a comparison of four different TAG parsing algorithms: the CYK-based algorithm used as an example in section 5.1, an Earley-based algorithm without the valid prefix property (described in Alonso et al. 1999, inspired in the one in Schabes 1994), an Earley-based algorithm with the valid prefix property (Alonso et al. 1999) and Nederhof's algorithm (Nederhof 1999). These parsers are compared on artificially generated grammars, by using our schema compiler to generate implementations and measuring their execution times with several grammars and sentences.

Note that the advantage of using artificially generated grammars is that we can easily see the influence of grammar size on performance. If we test the algorithms on grammars from real-life natural language corpora, as we did with the CFG parsers, we don't get a very precise idea of how the size of the grammar affects performance. Since our experience with CFG's showed this to be an important factor, and existing TAG parser performance comparisons (e.g. Díaz and Alonso 2000) work with a fixed (and small) grammar, we decided to use artificial grammars in order to be able to adjust both string size and grammar size in our experiments and see the influence of both factors.

For this purpose, given an integer $k > 0$, we define the tree-adjoining grammar G_k to be the grammar $G_k = (V_T, V_N, S, I, A)$ where $V_T = \{a_j | 0 \leq j \leq k\}$, $V_N = \{S, B\}$, and

$I = \{S(B(a_0))\}$[3],

$A = \{B(B(B^* \; a_j)) | 1 \leq j \leq k\}$.

Therefore, for a given k, G_k is a grammar with one initial tree and k auxiliary trees, which parses a language over an alphabet with $k + 1$ terminal symbols. The actual language defined by G_k is the regular language $L_k = a_0(a_1|a_2|..|a_k)^*$. [4] We shall note that although the languages L_k are trivial, the grammars G_k are built in such a way that any of the auxiliary trees may adjoin into any other. Therefore these grammars are suitable if we want to make an empyrical analysis of worst-case complexity.

Table 1 shows the execution time in milliseconds[5] of four TAG parsers with the grammars G_k, for different values of string length (n) and grammar size (k).

From this results, we can observe that both factors (string length and gram-

[3] Where trees are written in bracketed notation, and * is used to denote the foot node.

[4] Also, it is easy to prove that the grammar G_k is one of the minimal tree adjoining grammars (in terms of number of trees) whose associated language is L_k. Note that we need at least a tree containing a_0 as its only terminal in order to parse the sentence a_0, and for each $1 \leq i \leq k$, we need at least a tree containing a_i and no other a_j ($j > 0$) in order to parse the sentence $a_0 a_i$. Therefore, any TAG for the language L_k must have at least $k + 1$ elementary trees.

[5] The machine used for all the tests was an Intel Pentium 4 3.40 GHz, with 1 GB RAM and Sun Java Hotspot virtual machine (version 1.4.2_01-b06) running on Windows XP.

Runtimes in ms: Earley-based without the VPP					
String Size (n)	Grammar Size (k)				
	1	8	64	512	4096
2	~0	16	15	**1,156**	**109,843**
4	~0	31	63	**2,578**	**256,094**
8	16	31	**172**	**6,891**	**589,578**
16	31	172	**625**	**18,735**	**1,508,609**
32	110	609	**3,219**	**69,406**	
64	485	2,953	**22,453**	**289,984**	
128	2,031	**13,875**	**234,594**		
256	10,000	**101,219**			
512	61,266				

Runtimes in ms: CYK-based					
String Size (n)	Grammar Size (k)				
	1	8	64	512	4096
2	~0	~0	16	1,344	125,750
4	~0	~0	63	4,109	290,187
8	16	31	234	15,891	777,968
16	**15**	**62**	782	44,188	2,247,156
32	**94**	**312**	3,781	170,609	
64	**266**	**2,063**	25,094	550,016	
128	**1,187**	14,516	269,047		
256	**6,781**	108,297			
512	**52,000**				

Runtimes in ms: Nederhof's Algorithm					
String Size (n)	Grammar Size (k)				
	1	8	64	512	4096
2	~0	~0	47	1,875	151,532
4	~0	15	187	4,563	390,468
8	15	31	469	12,531	998,594
16	46	188	1,500	40,093	2,579,578
32	219	953	6,235	157,063	
64	1,078	4,735	35,860	620,047	
128	5,703	25,703	302,766		
256	37,125	159,609			
512	291,141				

Runtimes in ms: Earley-based with the VPP					
String Size (n)	Grammar Size (k)				
	1	8	64	512	4096
2	~0	~0	31	1,937	194,047
4	~0	16	78	4,078	453,203
8	15	31	234	10,922	781,141
16	31	188	875	27,125	1,787,140
32	125	750	4,141	98,829	
64	578	3,547	28,640	350,218	
128	2,453	20,766	264,500		
256	12,187	122,797			
512	74,046				

TABLE 1 Execution times of four different TAG parsers for artificially-generated grammars G_k. Best results are shown in boldface.

mar size) have an influence on runtime, and they interact between themselves: the growth rates with respect to one factor are influenced by the other factor, so it is hard to give precise estimates of empirical computational complexity. However, we can get rough estimates by focusing on cases where one of the factors takes high values and the other one takes low values (since in these cases the constant factors affecting complexity will be smaller) and test them by checking whether the sequence $T(n,k)/f(n)$ seems to converge to a positive constant for each fixed k (if $f(n)$ is an estimation of complexity with respect to string length) or whether $T(n,k)/f(k)$ seems to converge to a positive constant for each fixed n (if $f(k)$ is an estimation of complexity with respect to grammar size).

By applying these principles, we find that the empirical time complexity with respect to string length is in the range between $O(n^{2.8})$ and $O(n^3)$ for the CYK-based and Nederhof algorithms, and between $O(n^{2.6})$ and $O(n^3)$ for the Earley-based algorithms with and without the valid prefix property (VPP). Therefore, the practical time complexity we obtain is far below the theoretical worst-case bounds for these algorithms, which are $O(n^6)$ (except for the Earley-based algorithm with the VPP, which is $O(n^7)$).

Although for space reasons we don't include tables with the number of items generated in each case, our results show that the empirical space complexity with respect to string length is approximately $O(n^2)$ for all the algorithms, also far below the worst-case bounds ($O(n^4)$ and $O(n^5)$).

With respect to the size of the grammar, we obtain a time complexity of approximately $O(|I \cup A|^2)$ for all the algorithms. This matches the theoretical worst-case bound, which is $O(|I \cup A|^2)$ due to the adjunction steps, which work with pairs of trees. In the case of our artificially generated grammar, any auxiliary tree can adjoin into any other, so it's logical that our times grow quadratically. Note, however, that real-life grammars such as the XTAG English grammar (XTAG Research Group 2001) have relatively few different nonterminals in relation to their amount of trees, so many pairs of trees are susceptible of adjunction and we can't expect their behavior to be much better than this.

Space complexity with respect to grammar size is approximately $O(|I \cup A|)$ for all the algorithms. This is an expected result, since each generated item is associated to a given tree node.

Practical applications of TAG in natural language processing usually fall in the range of values for n and k covered in our experiments (grammars with hundreds or a few thousands of trees are used to parse sentences of several dozens of words). Within these ranges, both string length and grammar size take significant values and have an important influence on execution times, as we can see from the results in the tables. This leads us to note that traditional complexity analysis based on a single factor (string length or grammar size) can be misleading for practical applications, since it can lead us to an incomplete idea of real complexity. For example, if we are working with a grammar with thousands of trees, the size of the grammar is the most influential factor, and the use of filtering techniques (Schabes and Joshi 1991) to reduce the amount of trees used in parsing is essential in order to achieve good performance. The influence of string length in these cases, on the other hand, is mitigated by the huge constant factors related to grammar size. For instance, in the times shown in the tables for the grammar G_{4096}, we can see that parsing times are multiplied by a factor less than 3 when the length of the input string is duplicated, although the rest of the results have lead us to conclude that the practical asymptotic complexity with respect to string length is at least $O(n^{2.6})$. These interactions between both factors must be taken into account when analyzing performance in terms of computational complexity.

Earley-based algorithms achieve better execution times than the CYK-based algorithm for large grammars, although they are worse for small grammars. This contrasts with the results for context-free grammars, where CYK works better for large grammars: when working with CFG's, CYK has a better computational complexity than Earley (linear with respect to grammar size, see section 5.3), but the TAG variant of the CYK algorithm is quadratic with respect to grammar size and does not have this advantage.

CYK generates fewer items than the Earley-based algorithms when working with large grammars and short strings, and the opposite happens when working with small grammars and long strings.

The Earley-based algorithm with the VPP generates the same number of items than the one without this property, and has worse execution times. The reason is that no partial parses violating this property are generated by any of both algorithms in the particular case of this grammar, so guaranteeing the valid prefix property does not prevent any items from being generated. Therefore, the fact that the variant without the VPP works better in this particular case cannot be extrapolated to other grammars. However, the differences in times between these two algorithms illustrates the overhead caused by the extra checks needed to guarantee the valid prefix property in a particularly bad case.

Nederhof's algorithm has slower execution times than the other Earley variants. Despite the fact that Nederhof's algorithm is an improvement over the other Earley-based algorithm with the VPP in terms of computational complexity, the extra deductive steps it contains makes it slower in practice.

5.5 Parsing the XTAG English grammar

In order to complement our performance comparison of the four algorithms on artificial grammars, we have also studied the behavior of the parsers when working with a real-life, large-scale TAG: the XTAG English grammar (XTAG Research Group 2001).

The obtained execution times are in the ranges that we could expect given the artificial grammar results, i.e. they approximately match the times in the tables for the corresponding grammar sizes and input string lengths. The most noticeable difference is that the Earley-like algorithm verifying the valid prefix property generates fewer items that the variant without the VPP in the XTAG grammar, and this causes its runtimes to be faster. But this difference is not surprising, as explained in the previous section.

Note that, as the XTAG English grammar has over a thousand elementary trees, execution times are very large (over 100 seconds) when working with the full grammar, even with short sentences. However, when a tree selection filter is applied in order to work with only a subset of the grammar in function of the input string, the grammar size is reduced to one or two hundred trees and our parsers process short sentences in less than 5 seconds. Sarkar's XTAG distribution parser written in C[6] applies further filtering techniques and has specific optimizations for this grammar, obtaining better times for the XTAG than our generic parsers.

Table 2 contains a summary of the execution times obtained by our parsers for some sample sentences from the XTAG distribution. Note that the generated implementations used for these executions apply the mentioned tree filtering technique, so that the effective grammar size is different for each sentence, hence the high variability in execution times. More detailed information on these experiments with the XTAG English grammar can be found at (Gómez-Rodríguez et al., 2006a).

5.6 Overhead of TAG parsing over CFG parsing

The languages L_k that we parsed in section 5.4 were regular languages, so in practice we don't need tree adjoining grammars to parse them, although it was convenient to use them in our comparison. This can lead us to wonder how large is the overhead caused by using the TAG formalism to parse context-free languages.

Given the regular language $L_k = a_0(a_1|a_2|..|a_k)^*$, a context-free grammar that parses it is $G'_k = (N, \Sigma, P, S)$ with $N = \{S\}$ and

$$P = \{S \rightarrow a_0\} \cup \{S \rightarrow S\, a_i | 1 \leq i \leq k\}$$

This grammar minimizes the number of rules needed to parse L_k ($k + 1$ rules), but has left recursion. If we want to eliminate left recursion, we can use the grammar $G''_k = (N, \Sigma, P, S)$ with $N = \{S, A\}$ and

[6] Downloadable at: ftp://ftp.cis.upenn.edu/pub/xtag/lem/

Sentence	Runtimes in milliseconds			
	CYK	Ear. no VPP	Ear. VPP	Neder.
He was a cow	2985	**750**	**750**	2719
He loved himself	3109	1562	**1219**	6421
Go to your room	4078	1547	**1406**	6828
He is a real man	4266	1563	**1407**	4703
He was a real man	4234	1921	**1421**	4766
Who was at the door	4485	1813	**1562**	7782
He loved all cows	5469	2359	**2344**	11469
He called up her	7828	4906	**3563**	15532
He wanted to go to the city	10047	4422	**4016**	18969
That woman in the city contributed to this article	13641	**6515**	7172	31828
That people are not really amateurs at intelectual duelling	16500	**7781**	15235	56265
The index is intended to measure future economic performance	16875	17109	**9985**	39132
They expect him to cut costs throughout the organization	25859	**12000**	20828	63641
He will continue to place a huge burden on the city workers	54578	**35829**	57422	178875
He could have been simply being a jerk	**62157**	113532	109062	133515
A few fast food outlets are giving it a try	**269187**	3122860	3315359	

TABLE 2 Runtimes obtained by applying different XTAG parsers to several sentences. Best results for each sentence are shown in boldface.

$$P = \{S \to a_0 A\} \cup \{A \to a_i A | 1 \leq i \leq k\} \cup \{A \to \epsilon\}$$

which has $k + 2$ production rules.

The number of items generated by the Earley algorithm for context-free grammars when parsing a sentence of length n from the language L_k by using the grammar G'_k is $(k+2)n$. In the case of the grammar G''_k, the same algorithm generates $(k+4)n+\frac{n(n-1)}{2}+1$ items. In both cases the amount of items generated is linear with respect to grammar size, as in TAG parsers. With respect to string size, the amount of items is $O(n)$ for G'_k and $O(n^2)$ for G''_k, and it was approximately $O(n^2)$ for the TAG G_k. Note, however, that the constant factors behind complexity are much greater when working with G_k than with G''_k, and this reflects on the actual number of items generated (for example, the Earley algorithm generates 16,833 items when working with G''_{64} and a string of length $n = 128$, while the TAG variant of Earley without the valid prefix property generated 1,152,834 items).

The execution times for both algorithms appear in table 3. From the obtained times, we can deduce that the empirical time complexity is linear with respect to string length and quadratic with respect to grammar size in the case of G'_k; and quadratic with respect to string length and linear with respect to grammar size in the case of G''_k. So this example shows that, when parsing a context-free language using a tree-adjoining grammar, we get an overhead both in constant factors (more complex items, more deductive steps, etc.) and in asymptotic behavior, so actual execution times can be several orders of magnitude larger. Note that the way grammars are designed also has an influence, but our tree adjoining grammars G_k are the simplest TAGs able to parse the languages L_k by using adjunction (an alternative would be to write a grammar using the substitution operation to combine trees).

5.7 Conclusions

In this paper, we have presented a system that compiles parsing schemata to executable implementations of parsers, and used it to evaluate the performance of several TAG parsing algorithms, establishing comparisons both between themselves and with CFG parsers.

The results show that both string length and grammar size can be important factors in performance, and the interactions between them sometimes make their influence hard to quantify. The influence of string length in practical cases is usually below the theoretical worst-case bounds (between $O(n)$ and $O(n^2)$ in our tests for CFG's, and slightly below $O(n^3)$ for TAG's). Grammar size becomes the dominating factor in large TAG's, making tree filtering techniques advisable in order to achieve faster execution times.

Using TAG's to parse context-free languages causes an overhead both in

n	Grammar Size (k), grammar G'_k				
	1	8	64	512	4096
2	~0	~0	~0	31	2,062
4	~0	~0	~0	62	4,110
8	~0	~0	~0	125	8,265
16	~0	~0	~0	217	15,390
32	~0	~0	15	563	29,344
64	~0	~0	31	1,062	61,875
128	~0	~0	109	2,083	122,875
256	~0	15	188	4,266	236,688
512	15	31	328	8,406	484,859
n	Grammar Size (k), grammar G''_k				
	1	8	64	512	4096
2	~0	~0	~0	~0	47
4	~0	~0	~0	15	94
8	~0	~0	~0	16	203
16	~0	~0	~0	46	688
32	~0	~0	15	203	1,735
64	31	31	93	516	4,812
128	156	156	328	1,500	13,406
256	484	547	984	5,078	45,172
512	1,765	2,047	3,734	18,078	

TABLE 3 Runtimes obtained by applying the Earley parser for context-free grammars to sentences in L_k.

constant factors and in practical computational complexity, thus increasing execution times by several orders of magnitude with respect to CFG parsing.

Acknowledgements

The work reported in this article has been supported in part by Ministerio de Educación y Ciencia and FEDER (TIN2004-07246-C03-01, TIN2004-07246-C03-02), Xunta de Galicia (PGIDIT05PXIC30501PN, PGIDIT05PXIC10501PN, PGIDIT05SIN044E and PGIDIT05SIN059E), and Programa de becas FPU (Ministerio de Educación y Ciencia).

References

Alonso, Miguel A., David Cabrero, Eric de la Clergerie, and Manuel Vilares. 1999. Tabular algorithms for TAG parsing. In *Proc. of EACL'99, Ninth Conference of the European Chapter of the Association for Computational Linguistics*, pages 150–157. ACL, Bergen, Norway.

Carroll, J. 1993. Practical unification-based parsing of natural language. PhD thesis. Tech. Rep. 314, Computer Laboratory, University of Cambridge, Cambridge, UK.

Díaz, Víctor J. and Miguel A. Alonso. 2000. Comparing tabular parsers for tree adjoining grammars. In D. S. Warren, M. Vilares, L. Rodríguez Liñares, and M. A.

Alonso, eds., *Proc. of Tabulation in Parsing and Deduction (TAPD 2000)*, pages 91–100. Vigo, Spain.

Earley, J. 1970. An efficient context-free parsing algorithm. *Communications of the ACM* 13(2):94–102.

Gómez-Rodríguez, Carlos, Miguel A. Alonso, and Manuel Vilares. 2006a. Generating XTAG parsers from algebraic specifications. In *Proc. of TAG+8, the Eighth International Workshop on Tree Adjoining Grammar and Related Formalisms*. Sydney, Australia.

Gómez-Rodríguez, Carlos, Jesús Vilares, and Miguel A. Alonso. 2006b. Automatic generation of natural language parsers from declarative specifications. In *Proc. of STAIRS 2006*. Riva del Garda, Italy. Long version available at http://www.grupocole.org/GomVilAlo2006a_long.pdf.

Joshi, Aravind K. and Yves Schabes. 1997. Tree-adjoining grammars. In G. Rozenberg and A. Salomaa, eds., *Handbook of Formal Languages. Vol 3: Beyond Words*, chap. 2, pages 69–123. Berlin/Heidelberg/New York: Springer-Verlag.

Kasami, T. 1965. An efficient recognition and syntax algorithm for context-free languages. Scientific Report AFCRL-65-758, Air Force Cambridge Research Lab., Bedford, Massachussetts.

Nederhof, Mark-Jan. 1999. The computational complexity of the correct-prefix property for TAGs. *Computational Linguistics* 25(3):345–360.

Rosenkrantz, D. J. and P. M. Lewis II. 1970. Deterministic Left Corner parsing. In *Conference Record of 1970 Eleventh Annual Meeting on Switching and Automata Theory*, pages 139–152. IEEE, Santa Monica, CA, USA.

Sampson, G. 1994. The Susanne corpus, release 3.

Schabes, Yves. 1994. Left to right parsing of lexicalized tree-adjoining grammars. *Computational Intelligence* 10(4):506–515.

Schabes, Yves and Aravind K. Joshi. 1991. Parsing with lexicalized tree adjoining grammar. In M. Tomita, ed., *Current Issues in Parsing Technologies*, chap. 3, pages 25–47. Norwell, MA, USA: Kluwer Academic Publishers. ISBN 0-7923-9131-4.

Schoorl, J. J. and S. Belder. 1990. Computational linguistics at Delft: A status report, Report WTM/TT 90–09.

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

Sikkel, Klaas. 1997. *Parsing Schemata — A Framework for Specification and Analysis of Parsing Algorithms*. Texts in Theoretical Computer Science — An EATCS Series. Berlin/Heidelberg/New York: Springer-Verlag. ISBN 3-540-61650-0.

Vijay-Shanker, K. and Aravind K. Joshi. 1985. Some computational properties of tree adjoining grammars. In *23rd Annual Meeting of the Association for Computational Linguistics*, pages 82–93. ACL, Chicago, IL, USA.

XTAG Research Group. 2001. A lexicalized tree adjoining grammar for English. Tech. Rep. IRCS-01-03, IRCS, University of Pennsylvania.

Younger, D. H. 1967. Recognition and parsing of context-free languages in time n^3. *Information and Control* 10(2):189–208.

6

Properties of Binary Transitive Closure Logic over Trees

STEPHAN KEPSER

Abstract

Binary transitive closure logic (FO* for short) is the extension of first-order predicate logic by a transitive closure operator of binary relations. It is known that this logic is more powerful than FO on arbitrary structures and on finite ordered trees. It is also known that it is at most as powerful as monadic second-order logic (MSO) on arbitrary structures and on finite trees. We will study the expressive power of FO* on trees to show that several MSO properties can be expressed in FO*.

The following results will be shown.

- A linear order can be defined on trees.
- The class EVEN of trees with an even number of nodes can be defined.
- On arbitrary structures with a tree signature, the classes of trees and finite trees can be defined.
- FO* is strictly more powerful than tree walking automata.

These results imply that FO* is neither compact nor does it have the Löwenheim-Skolem-Upward property.

6.1 Introduction

The question about the best suited logic for describing tree properties or defining tree languages is an important one for model theoretic syntax as well as for querying treebanks. Model theoretic syntax is a research program in mathematical linguistics concerned with studying the descriptive complexity of grammar formalisms for natural languages by defining their derivation trees in suitable logical formalisms. Since the very influential book by Rogers

FG-2006.
Edited by Paola Monachesi, Gerald Penn,
Giorgio Satta and Shuly Wintner.

(1998) it is monadic second-order logic (MSO) or even more powerful logics that are used to describe linguistic structures.

With the advent of XML and query languages for XML documents, in particular XPath, the interest in logics for querying treebanks rose dramatically. There is now a large interest in this topic in computer science. Independent of that, but temporarily parallel, large syntactically annotated treebanks became available in linguistics. They provide nowadays a rich and important source for the study of language. But in order to access this source, suitable query languages for treebanks are required.

One of the simplest properties that are known to be inexpressible in first-order predicate logic (FO henceforth) is the transitive closure of a binary relation. It is therefore a natural move to extend FO by a binary transitive closure operator. And this move has been done before in the definition of query languages for relational databases, in particular for the SQL3 standard. But it seems that the expressive power of FO plus binary transitive closures (FO^* for short) to define tree properties is not much studied yet. This is somewhat surprising, because there is reason to believe that FO^* is more user friendly than MSO. Most users of query languages, in particular linguists, understand the concept of a transitive closure very well and know how to use it. It is a lot more difficult to use set variables to describe tree properties. An example for this claim is the fact MSO is capable of defining binary transitive closures, as shown by Moschovakis (1974). His formula is given at the end of the next section. It is questionable that ordinary users (without profound knowledge of MSO) would be able to find this formula.

We propose to seriously consider FO^* as a language for defining tree properties. We do so by showing that several important MSO definable properties can be defined in FO^*. One such example is the ability to define a linear order on the nodes of a tree. The order resembles depth-first left-to-right traversal of a tree. A linear order is a powerful concept that can be used defining additional properties. For example, it is used to count the number of nodes in a tree modulo a given natural number. An instance is the definition of the class EVEN of all trees with an even number of nodes in FO^*.

Arguably an important reason for Rogers' choice of MSO is its ability to axiomatise trees. I.e., there exists a set of axioms such that an arbitrary structure (of a suitable signature) is a tree – finite or infinite – iff it is a model of the axioms. It is known that this characterisation of trees cannot be done using FO. But the full expressive power of MSO may not really be needed for the axiomatisation, because we show that arbitrary trees and finite trees can be axiomatised in FO^*. This capability of axiomatising finite and infinite trees implies that FO^* is neither compact nor does it possess the Löwenheim-Skolem-Upward property.

There exists a tree automaton concept that defines serial instead of parallel processing of trees, namely tree walking automata (TWA). As the name implies, a tree is processed by walking up and down in the tree and inspecting nodes serially. One may therefore believe that these automata could be the automaton-theoretic correspondant of FO*. But we show here that FO* is more powerful. Every tree language that is recognised by a TWA can be defined in FO*. But there are FO*-definable tree languages that cannot be recognised by any TWA.

6.2 Preliminaries

Let M be a set. We write $\wp(M)$ for the power set of M. Let $R \subseteq M \times M$ be a binary relation over M. The *transitive closure* $TC(R)$ of R is the smallest set containing R and for all $x, y, z \in M$ such that $(x, y) \in TC(R)$ and $(y, z) \in TC(R)$ we have $(x, z) \in TC(R)$. I.e.,

$$TC(R) := \bigcap \{W \mid \begin{array}{l} R \subseteq W \subseteq M \times M, \forall x, y, z \in M : \\ (x, y), (y, z) \in W \implies (x, z) \in W\}. \end{array}$$

We consider labelled ordered unranked trees. A tree is ordered if the set of child nodes of every node is linearly ordered. A tree is unranked if there is no relationship between the label of a node and the number of its children. In Sections 6.3 and 6.5 we only consider finite trees, in Section 6.4 we also consider infinite trees.

Definition 1 A *tree domain* is a non-empty subset $T \subseteq \mathbb{N}^*$ such that for all $u, v \in \mathbb{N}^* : uv \in T \implies u \in T$ (closure under prefixes) and for all $u \in \mathbb{N}^*$ and $i \in \mathbb{N} : ui \in T \implies uj \in T$ for all $j < i$ (closure under left sisters).

Let $\mathcal{L}$ be a set of labels. A *tree* is a pair (T, Lab) where T is a tree domain and $Lab : T \to \mathcal{L}$ is a node labelling function.

A tree is *finite* iff its tree domain is finite.

We remark that a tree domain is at most countable, since it is a subset of a countable union of countable sets.

The language to talk about trees will be an extension of first-order logic. Its syntax is as follows. Let $X = \{x, y, z, w, u, x_1, x_2, x_3, \ldots\}$ be a denumerable infinite set of variables. The atomic formulae are $L(x)$ for each label $L \in \mathcal{L}$, $x \to y$, and $x \downarrow y$. Complex formulae are constructed from simpler ones by means of the boolean connectives, existential and universal quantification, and transitive closure. I.e., if ϕ and ψ are formulae, then $\neg\phi$, $\phi \wedge \psi$, $\phi \vee \psi$, $\exists x$: ϕ, $\forall x$:ϕ, and $[\mathrm{TC}_{x_1,x_2}\ \phi](x, y)$ are formulae.

The semantics of the first-order part of the language is standard. Let (T, Lab) be a tree. A variable assignment $a : X \to T$ assigns variables to nodes in the tree. The root node has the empty address ϵ. Now $[\![L(x)]\!]^a = \mathrm{T}$ iff $Lab(a(x)) = L$. $[\![x \downarrow y]\!]^a = T$ iff $a(y) = a(x)i$ for some $i \in \mathbb{N}$, i.e., $\downarrow$ is the parent relation. $[\![x \to y]\!]^a = \mathrm{T}$ iff there is a $u \in T$ and $i \in \mathbb{N}$ such that

$a(x) = ui$ and $a(y) = ui + 1$, i.e., $\rightarrow$ is the immediate sister relation.

Boolean connectives and quantification have their standard interpretation. Now, $[\![[\mathrm{TC}_{x_1,x_2}\ \phi](x,y)]\!]^a = \mathrm{T}$ iff

$$(a(x), a(y)) \in TC(\{(b,d) \mid [\![\phi]\!]^{ab/x_1 d/x_2} = \mathrm{T}\})$$

where $ab/x_1d/x_2$ is the variable assignment that is identical to a except that x_1 is assigned to b and x_2 to d. If ϕ is a formula with free variables x_1, x_2, it can be regarded as a binary relation $\phi(x_1, x_2)$. Then $[\mathrm{TC}_{x_1,x_2}\ \phi]$ is the transitive closure of this binary relation. This language is abbreviated FO*.

FO* is amongst the smallest extension of first-order logic. It is known that the transitive closure of a binary relation is *not* first-order definable (Fagin, 1975). But when talking about trees, people frequently want to talk about paths in a tree. And a path is the transitive closure of certain base steps. FO* has at most the expressive power of monadic second-order logic (MSO). It is an old result, which goes back at least to Moschovakis (1974, p. 20), that the transitive closure of every MSO-definable binary relation is also MSO-definable. Let R be an MSO-definable binary relation. Then

$$\begin{aligned}\forall X\ \ &(\forall z, w(z \in X \land R(z,w) \implies w \in X) \land \forall z(R(x,z) \implies z \in X))\\ &\implies y \in X\end{aligned}$$

is a formula with free variables x and y that defines the transitive closure of R. It follows that every tree language definable in FO* can be defined in MSO.

6.3 Definability of Order

One of the abstract insights from descriptive complexity theory is that order is a very important property of structures. The relationship between certain logics and classical complexity classes is frequently restricted to *ordered* structures, i.e., structures where the carrier is linearly ordered. The reason for this restriction is to be found in the fact that computation is an ordered process. Definability and non-definability results for certain logics over ordered structures frequently do not extend to unordered structures. It is therefore an important property of a logic, if the logic itself is capable of expressing order without recourse to an extended signature. The probably best known logic with this property is Σ^1_1, the extension of first-order logic by arbitrary relation variables that are existentially quantified. It is obviously possible to define order in Σ^1_1, because we can say there is a binary relation that has all the properties of a linear order. These properties are known to be first-order properties. It is hence the ability to say "there is a binary relation" that is the key.

There is no way that FO* could define order on arbitrary finite structures. But if we only consider trees as models, FO* can define order. Indeed it is possible to give a definition of the depth-first left-to-right order of nodes in a tree (and some variants).

Proposition 9 *There is an explicit definition of a linear order of the nodes in a tree in FO*.*

Proof. Define the proper dominance relation of trees $Dom(x, y)$ as $[\mathrm{TC}_{x,y}\ x \downarrow y](x, y)$. Similarly, define the sister relation $Sis(x, y)$ as $[\mathrm{TC}_{x,y}\ x \rightarrow y](x, y)$. Now define $x < y$ as

$$\begin{aligned} Dom(x, y) \vee (\exists w, v :\ & Sis(w, v) \wedge \\ & (w = x \vee Dom(w, x)) \wedge (v = y \vee Dom(v, y))). \end{aligned}$$

The first disjunct expresses the "depth-first" part of the order. The more complicated second disjunct formalises the "left-to-right" part. It expresses that there is a common ancestor of nodes x and y and node x is to be found on a left branch while y is to be found on a right branch. Care is taken that mutual domination is excluded. Hence the two disjuncts are mutually exclusive. Since the dominance and the sisterhood steps are both irreflexive, the whole relation $<$ is irreflexive. Furthermore for each pair of distinct nodes in a tree, either one dominates the other, or there is a common ancestor such that one node is on a left branch while the other is on a right branch. Hence the relation is total. Transitivity can easily be checked by considering the four cases involved in expanding $x < y$ and $y < z$. □

Note that the root node is the smallest element of the order. If the tree is finite, the largest element is the leaf of the rightmost branch of the tree. The root node is FO-definable via $\neg\exists y : y \downarrow x$. The largest element Max of the order is FO*-definable by $\exists x \neg\exists y : x < y$. The successor y of a node x in the linear order ($Succ(x, y)$) is also FO*-definable: $x < y \wedge \neg\exists z : x < z \wedge z < y$. Using a linear order it is possible to do modulo counting on trees. That is for $n, k \in \mathbb{N}$ we can define the class of finite trees such that each tree in the class has $d \times n + k$ nodes (for some $d \in \mathbb{N}$). As an example, we define the class EVEN of trees with an even number of nodes (i.e, $n = 2, k = 0$).

Proposition 10 *The class of finite trees with an even number of nodes is FO*-definable.*

Proof. We only consider the case where a tree has more than two nodes. The formula

$$\exists w : Succ(Root, w) \wedge [TC_{x,y} \exists z : Succ(x, z) \wedge Succ(z, y)](w, Max)$$

expresses that we go in one step from the root to its successor w. From w we can reach the last element of the order by an arbitrary number of two successor steps. If we take the two-successors-step path through the linear order from the root to the maximum, we have an odd number of nodes, since a path of n double-successor-steps has $n + 1$ nodes. □

Corollary 11 *FO* has* no *normal form of the type* $[\mathrm{TC}_{x,y}\ \phi(x, y)](r, r)$ *where* $\phi(x, y)$ *is an FO formula and r the root.*

Proof. With a single application of a TC-operator we can define trees with a linear order. If FO with a single TC-operator is interpreted over finite successor structures, then it is equivalent to FO with order. But over finite orderings, EVEN is not definable in FO. □

6.4 Definability of Tree Structures

In previous and all following sections we assume that we only consider tree models as defined in the preliminaries section. But in this section we take a more general view, a view that has its origin in model theoretic syntax. The aim is to find whether it is possible to give an axiomatisation of those structures linguists are interested in. This task has two subparts. The first consists of defining trees, or more precisely finite trees, as the intended models. The second part consists of axiomatising linguistic principles such as the Binding theory in the given logic. We will only be concerned with the first part here. This section is inspired by the book by Rogers (1998). More specifically we show that the main results of Chapter 3 carry over to FO^*. We will frequently cite this chapter in the current section.

The language of this section is binary transitive closure logic with equality over the following base relations:

$\triangleleft$	parent relation
$\triangleleft^*$	dominance relation
$\triangleleft^+$	proper dominance relation
$\prec$	left-of relation

We also assume there to be a set $\mathcal{L}$ of unary predicate symbols representing linguistic labels. We write $FO^{*\triangleleft}$ for this language to indicate that the base relations differ from the ones in the other sections of this paper.

A model for $FO^{*\triangleleft}$ is a tuple (U, P, D, L, Lab) where U is a non-empty domain, P, D and L are binary relations over U interpreting $\triangleleft, \triangleleft^*$ and $\prec$. And $Lab : \mathcal{L} \to \wp(U)$ interpretes each label as a subset of U.

Since the intended models of this language are trees, we have to restrict the class of models by giving axioms of trees. Many properties of trees can be defined by first-order axioms. The following 12 axioms are cited from (Rogers, 1998, p. 15f.).

A1 $\exists x \forall y : x \triangleleft^* y$
(Connectivity wrt dominance)

A2 $\forall x, y : (x \triangleleft^* y \wedge y \triangleleft^* x) \to x = y$
(Antisymmetry of dominance)

A3 $\forall x, y, z : (x \triangleleft^* y \wedge y \triangleleft^* z) \to x \triangleleft^* z$
(Transitivity of dominance)

A4 $\forall x, y : x \triangleleft^+ y \leftrightarrow (x \triangleleft^* y \wedge x \neq y)$
(Definition of proper dominance)

A5 $\forall x, y : x \triangleleft y \leftrightarrow (x \triangleleft^{+} y \wedge \forall z : (x \triangleleft^{*} z \wedge z \triangleleft^{*} y) \rightarrow (z \triangleleft^{*} x \vee y \triangleleft^{*} z))$
(Definition of immediate dominance)

A6 $\forall x, z : x \triangleleft^{+} z \rightarrow ((\exists y : x \triangleleft y \wedge y \triangleleft^{*} z) \wedge (\exists y : y \triangleleft z))$
(Discreteness of dominance)

A7 $\forall x, y : (x \triangleleft^{*} y \wedge y \triangleleft^{*} x) \leftrightarrow (x \nprec y \wedge y \nprec x)$
(Exhaustiveness and exclusiveness)

A8 $\forall w, x, y, z : (x \prec y \wedge x \triangleleft^{*} w \wedge y \triangleleft^{*} z) \rightarrow w \prec z$
(Inheritance of Left-of wrt dominance)

A9 $\forall x, y, z : (x \prec y \wedge y \prec z) \rightarrow x \prec z$
(Transitivity of left-of)

A10 $\forall x, y : x \prec y \rightarrow y \nprec x$
(Asymmetry of left-of)

A11 $\forall x (\exists y : x \triangleleft y) \rightarrow (\exists y : x \triangleleft y \wedge \forall z : x \triangleleft z \rightarrow z \nprec y)$
(Existence of a minimum child)

A12 $\forall x, z : x \prec z \rightarrow \quad (\exists y : x \prec y \wedge \forall w : x \prec w \rightarrow w \nprec y) \wedge$
$\qquad (\exists y : y \prec z \wedge \forall w : w \prec z \rightarrow y \nprec w)$
(Discreteness of left-of)

A discussion of these axioms can be found in (Rogers, 1998, p. 16f.). Every tree (finite or infinite) obeys to these axioms. But there are non-standard models, i.e., structures that are models of theses axioms but would not be considered as trees. Actually, it is *not* possible to give a first-order axiomatisation of trees, as was shown by Backofen et al. (1995). A look at the non-standard model given by Backofen et al. (1995) helps to understand where the problem is located. Consider the model M of Figure 1. It consists of two components: an infinite sequence of nodes, each with a single child, extending up from the root; and, infinitely far out, a second component in which every node has exactly two children, every node has a parent in that component, and every node is dominated by every node in the first component. The arrows in the figure are intended to suggest that there is no maximal point (wrt dominance) among the set of points with single children and no minimal point (wrt dominance) among the set of points with two children.

It is easy to see that the proper dominance relation does not only contain the immediate dominance relation but also the transitive closure of the immediate dominance. In the nonstandard models, proper dominance truly extends the transitive closure of immediate dominance. In the example, all nodes of the first component properly dominate all nodes of the second component. But this part of the dominance relation is not contained in the transitive closure of immediate dominance. In a proper tree model, the proper dominance is always identical to the transitive closure of immediate dominance. This insight can be expressed in $\mathrm{FO}^{*}{\triangleleft}$ as an axiom.

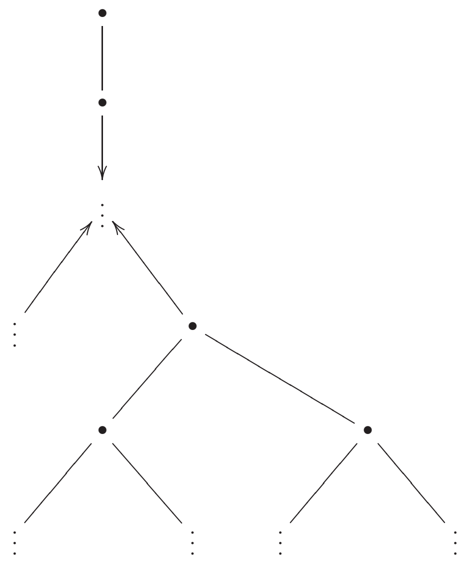


FIGURE 1 A nonstandard model of the first-order tree axioms.

AT1 $\forall x, y : x \triangleleft^+ y \rightarrow [\mathrm{TC}_{z,w} z \triangleleft w](x, y)$
(Proper dominance is the transitive closure of immediate dominance)

Another way of reading this axiom is to say that the path from an arbitrary node back to the root is finite. AT1 together with the first-order axioms does still not suffice to axiomatise proper trees. Consider the sisters of a node. They are ordered by $\prec$, and there is a left-most sister. Now, in a proper tree, the number of sisters to the left is finite for every node. This can be axiomatised as follows. We can easily define that one node is the immediate sister of another node. The relation $IS(x, y)$ is defined as $\exists z : z \triangleleft x \wedge z \triangleleft y \wedge x \prec y \wedge \neg\exists w : x \prec w \prec y$. Now we can spell out an axiom analogue to AT1.

AT2 $\forall x, y, z : (x \triangleleft y \wedge x \triangleleft z \wedge y \prec z) \rightarrow [\mathrm{TC}_{v,w} IS(v, w)](y, z)$
(Finitely many left sisters)

Theorem 12 *Axioms A1–A12, AT1, and AT2 define the class of tree models.*

Proof. The proof is analogous to the proof of Theorem 3.9 in (Rogers, 1998). Consider in particular Footnote 8 on page 23.

Rogers showed that every tree (in the sense of Definition 1) is a model of axioms A1–A12 and for each node $x \in U$ the sets $A_x = \{(y, x) \in D\}$ of ancestors of x and $L_x = \{y \mid \exists z : (z, x), (z, y) \in D \text{ and } (y, x) \in L\}$ of left sisters of x are finite (Lemma 3.5). And every tree obviously satisfies axioms

AT1 and AT2. Furthermore, each model of axioms A1–A12 where A_x and L_x are finite for each node $x \in U$ is isomorphic to a tree (Lemma 3.6). Now suppose a model of A1–A12 satisfies AT1. Then for each node $x \in U$ the set A_x is finite, because it contains the root (A1) and is constructed of parent-child steps (AT1), and a transitive closure of single steps cannot reach a limit ordinal. An analogous argument can be made with respect to models of A1–A12 and AT2. Hence for every model of of A1–A12, AT1, and AT2 and all nodes $x \in U$ we see that the sets A_x and L_x are finite. By the above quoted Lemma 3.6, these models are isomorphic to trees. ☐

The tree models of Axioms A1–A2, AT1, and AT2 can be finite as well as infinite. But since they are all tree models, they are at most countable. This is because every tree domain is at most countable (see remark after Definition 1). And every tree model is isomorphic to a tree. As an immediate consequence we get that FO* does *not* have the Löwenheim-Skolem-Upward property. This property states that if a theory (i.e., potentially infinite set of sentences) has a model of size ω it has models of arbitrary infinite cardinalities.

Linguists are mostly (if not exclusively) concerned with finite trees. Hence it would be nice if we could restrict the class of models further down to finite trees. This can indeed be done. Rogers (1998) defines a linear order on the nodes of a tree as follows. Node $x < y$ iff $x \triangleleft^+ y \lor x \prec y$. By Axiom A7, each pair of nodes is either a member of the dominance relation or a member of the left-of relation. Hence this defines indeed a linear order. Actually, the order is the same as the one in the previous section: depth-first left-to-right tree traversal. As in the previous section we use $Succ(x, y)$ for y being the immediate successor of x in the order. Finiteness can now be defined in two steps. Firstly we demand the linear order to be the transitive closure of the immediate successor relation. The consequence of this demand is that for every element in the order there is only a finite number of nodes that are smaller than this element. Secondly we demand the order to have a maximal element. If the maximal element has only a finite number of elements smaller than it, the tree is obviously finite.

AF $\forall x, y : x < y \implies [\mathrm{TC}_{x,y}\, Succ(x, y)](x, y) \quad \land$
$\exists x \forall y : y < x \lor y = x.$
(Finiteness of the order <)

Theorem 13 *Axioms A1–A12, AT1, AT2, and AF define the class of* finite *tree models.*

Proof. By Theorem 12, every model of the Axioms A1–A12, AT1, and AT2 is isomorphic to a tree model. If a model is finite, then AF is certainly true. For the converse, assume that $\forall x, y : x < y \implies [\mathrm{TC}_{x,y}\, Succ(x, y)](x, y)$. By definition of the TC-operator, the set $\{y \mid y < x\}$ of elements smaller than x is finite for every node x. If the order has additionally a maximal element m, then it is finite. □

As a simple consequence of the above theorem we get that FO^* is *not* compact.

6.5 FO* and Tree Walking Automata

Tree walking automata were introduced by Aho and Ullman (1971) as sequential automata on trees. At every moment of its run, a TWA is in a single node of the tree and in one of a finite number of states. It walks around the tree choosing a neighboring node based on the current state, the label of the current node, and the child number of the current node.

More formally, we consider trees of maximal branching degree k. The following definition is mainly cited from (Bojanczyk and Colcombet, 2005). Every node v has a type. The possible values are Types $= \{r, 1, 2, \ldots, k\} \times \{l, i\}$ where r stands for the root, $j \in \{1, \ldots, k\}$ states that v is the j-th child, l states that v is a leaf, i that v is an internal node. A direction is an element of Dir $= \{\uparrow, \downarrow_1, \ldots, \downarrow_k, \mathit{stay}\}$ where $\uparrow$ stands for 'move to the parent', $\downarrow_j$ 'move to the j-th child, and *stay* to 'stay at the current node'. A TWA is a quintuple $(S, \Sigma, \delta, s_0, F)$ where S is a finite set of states, Σ is the alphabet of node labels, $s_0 \in S$ is the initial state and $F \subseteq S$ is the set of final states. The transition relation δ is of the form

$$\delta \subseteq S \times \text{Types} \times \Sigma \times S \times \text{Dir}.$$

A configuration is a pair of a node and a state. A run is a sequence of configurations where every two consecutive configurations are consistent with the transition relation. A run is accepting iff it starts and ends at the root of the tree, the first state is s_0 and the last state is a member of F. The TWA accepts a tree iff there is an accepting run. The set of Σ-trees recognised by a TWA is the set of trees for which there is an accepting run.

Bojanczyk and Colcombet (2005) showed that TWA cannot recognise all regular tree languages. This means that MSO and tree automata are strictly more powerful than TWA. In an extension of their proof we will show that even FO^* is more powerful than TWA.

Theorem 14 *The classes of tree languages definable in FO^* strictly extend the classes of tree languages recognisable by TWA.*

Proof. The proof consists of two parts. We will first show that every TWA-recognisable tree language is FO^*-definable. Secondly we will show that there is an FO^*-definable tree language that cannot be recognised by any TWA.

The first part of the proof is based on recent results by Neven and Schwentick (2003). They showed that a tree language is recognisable by a TWA if and only if it is definable by a formula of the following type: $[\mathrm{TC}_{x,y}\ \phi(x, y)](r, r)$ where r is a constant for the root of a tree, ϕ is an FO formula with additional unary $depth_m$ predicates. Apart from the $depth_m$ predicates, these formulae are obviously in FO*. Now, $depth_m(x)$ is true iff x is a multiple of m steps away from the root. For every m, the predicate $depth_m$ can be defined by an FO*-formula: $[\mathrm{TC}_{x_0,x_m}\ \exists x_1, \ldots x_{m-1} : x_0 \downarrow x_1 \wedge \cdots \wedge x_{m-1} \downarrow x_m](r, x)$ is a predicate that is true on a node x just in case there is a $k \in \mathbb{N}$ such that x is at depth $k \times m$. Thus every TWA-recognisable tree language is FO*-definable.

To show the second half of the theorem, we will indicate that the separating language L given by Bojanczyk and Colcombet (2005) can be defined in FO*. The authors consider binary trees. They show (in Fact 1) that L can be defined in first-order logic with the following three basic relations: left and right child, and ancestor relation. Now, left and right child are obviously FO*-definable relations. And the ancestor relation is easily FO*-definable: $[\mathrm{TC}_{x,y}\ x \downarrow y]$. ☐

The relationship between TWA and transitive closure logics was recently also studied by Engelfriet and Hoogeboom (2006). They show that if one extends deterministic TWA by finite sets of pebbles, they have the same expressive power as deterministic transitive closure logics.

6.6 Conclusion

We showed a number of properties of FO* to indicate that it should seriously be considered as a logic for defining tree languages. Although the addition of binary transitive closure to first-order logic can be seen as a small one, FO* is capable of expressing important second-order properties over trees. It is possible to define a linear order over the nodes in a tree. And using this order one can count modulo any natural number. On arbitrary structures with appropriate signature one can axiomatise the classes of trees and finite trees. These axiomatisations showed that FO* is neither compact nor does it have the Löwenheim-Skolem-Upward property. Furthermore although tree walking automata look like they might serve as an automaton model for FO*, it turns out that FO* is more powerful than TWA.

A word about complexity issues may be in place. FO* has quite a good data complexity. By translating FO* formulae into MSO formulae and using the equivalence between MSO and tree automata one can see that FO* has a linear time data complexity. And since FO* is a sublogic of FO+TC (see below), it also has NLOGSPACE data complexity. A straight-forward implementation of transitive closure yields a PTIME query complexity. It is unclear to the author whether this result can be improved upon.

The main open question is of course whether FO* is strictly less powerful than MSO. It is also interesting to study the relationship of FO* to modal languages for trees like PDL$_{Tree}$ (Kracht, 1995). Marx (2004) basically showed that PDL_{Tree} is at most as powerful as FO^{*3}, where FO^{*3} is the restriction of FO^{*} where every formula has at most 3 different variables. ten Cate (2006) recently showed that queries in XPath with Kleene star and loop predicate have the same expressive power as FO^{*3}.

One may also ask what happens if we introduce the transitive closure of arbitrary relations, not just binary ones. This logic (abbreviated FO+TC) was introduced by Immerman (see Immerman, 1999) to logically describe the complexity class NLOGSPACE. Tiede and Kepser (2006) have recently shown that FO+TC is more expressive than MSO over trees. The statement remains true even if one only considers *deterministic* transitive closures.

Acknowledgements

The author wishes to thank four anonymous referees whose comments helped improving the quality of the paper.
This research was funded by a grant of the German Research Foundation (DFG SFB-441).

Stephan Kepser
Collaborative Research Centre 441
University of Tübingen
Germany

References

Aho, Alfred V. and Jeffrey D. Ullman. 1971. Translations on a context-free grammar. *Information and Control* 19:439–475.

Backofen, Rolf, James Rogers, and Krishnamurti Vijay-Shanker. 1995. A first-order axiomatization of the theory of finite trees. *Journal of Logic, Language, and Information* 4:5–39.

Bojanczyk, Mikolaj and Thomas Colcombet. 2005. Tree-walking automata do not recognize all regular languages. In H. N. Gabow and R. Fagin, eds., *The 37th ACM Symposium on Theory of Computing (STOC 2005)*, pages 234–243. ACM.

Engelfriet, Joost and Hendrik Jan Hoogeboom. 2006. Nested pebbles and transitive closure. In B. Durand and W. Thomas, eds., *STACS 2006*, vol. LNCS 3884, pages 477–488. Springer.

Fagin, Ronald. 1975. Monadic generalized spectra. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik* 21:89–96.

Immerman, Neil. 1999. *Descriptive Complexity*. Springer.

Kracht, Marcus. 1995. Syntactic codes and grammar refinement. *Journal of Logic, Language, and Information* 4(1):41–60.

Marx, Maarten. 2004. XPath with conditional axis relations. In E. Bertino, S. Christodoulakis, D. Plexousakis, V. Christophides, M. Koubarakis, K. Böhm, and E. Ferrari, eds., *Advances in Database Technology – EDBT 2004*, vol. LNCS 2992, pages 477–494. Springer.

Moschovakis, Yiannis. 1974. *Elementary Induction on Abstract Structures*. North-Holland Publishing Company.

Neven, Frank and Thomas Schwentick. 2003. On the power of tree-walking automata. *Information and Computation* 183(1):86–103.

Rogers, James. 1998. *A Descriptive Approach to Language-Theoretic Complexity*. CSLI Publications.

ten Cate, Balder. 2006. Expressivity of XPath with transitive closure. In J. van den Bussche, ed., *Proceedings of PODS 2006*, pages 328–337.

Tiede, Hans-Jörg and Stephan Kepser. 2006. Monadic second-order logic over trees and transitive closure logics. In G. Mints, ed., *3th Workshop on Logic, Language, Information, and Computation*.

7

Pregroups with modalities

Aleksandra Kislak-Malinowska

Abstract

In this paper we concentrate mainly on the notion of β-pregroups, which are pregroups (first introduced by Lambek Lambek (1999) in 1999) enriched with modality operators. β-pregroups were first proposed by Fadda Fadda (2002) in 2001. The motivation to introduce them was to limit (locally) the associativity in the calculus considered. In this paper we present this new calculus in the form of a rewriting system, prove the very important feature of this system - that in a given derivation the non-expanding rules must always proceed non-contracting ones in order the derivation to be minimal (normalization theorem). We also propose a sequent system for this calculus and prove the cut elimination theorem for it.

Keywords Pregroup, β-pregroup, normalization theorem, cut elimination.

7.1 Introduction

Definition 2 A pregroup is a structure $(G, \leq, \cdot, l, r, 1)$ such that $(G, \leq, \cdot, 1)$ is a partially ordered monoid, and l, r are unary operations on G, fulfilling the following conditions:

$$a^l a \leq 1 \leq aa^l \text{ and } aa^r \leq 1 \leq a^r a \tag{7.1}$$

for all $a \in G$. Element a^l (a^r respectively) is called the left (right) adjoint of a.

The notion of a pregroup, introduced by Lambek Lambek (1999), is connected to the notion of a residuated monoid, known from the theory of partially ordered algebraic systems.

Theorem 15 *Lambek (1999) In each pregroup the following equalities and inequalities are valid:*

FG-2006.
Edited by Paola Monachesi, Gerald Penn,
Giorgio Satta and Shuly Wintner.

$$1^l = 1^r = 1, \quad a^{lr} = a = a^{rl}, \tag{7.2}$$

$$(ab)^l = b^l a^l, \quad (ab)^r = b^r a^r, \tag{7.3}$$

$$a \leq b \quad \textit{iff} \quad b^l \leq a^l \quad \textit{iff} \quad b^r \leq a^r. \tag{7.4}$$

For any arbitrary element a of a pregroup we define an element $a^{(n)}$, for $n \in Z$, in a following way:
$a^0 = a$,
$a^{(n+1)} = (a^{(n)})^r$,
$a^{(n-1)} = (a^{(n)})^l$.

As a consequence of (2) and (7.4) we get:

$$a^{(n)} a^{(n+1)} \leq 1 \leq a^{(n+1)} a^{(n)} \tag{7.5}$$

$$if\ a \leq b\ then\ a^{(2n)} \leq b^{(2n)}\ and\ b^{(2n+1)} \leq a^{(2n+1)} \tag{7.6}$$

for all $n \in Z$.

Let $(P, \leq)$ be a poset. Elements of the set P are treated as constants. *Terms* are expressions of the form $p^{(n)}$, for $p \in P$, $n \in Z$; $p^{(0)}$ is equal p. *Types* are finite strings of terms, denoted by X, Y, Z, V, U etc.

The basic rewriting rules are as follows:

(CON) - contraction:
$X, p^{(n)}, p^{(n+1)}, Y \to X, Y$;

(EXP) - expansion:
$X, Y \to X, p^{(n+1)}, p^{(n)}, Y$;

(IND) - induced step:
$X, p^{(2n)}, Y \to X, q^{(2n)}, Y$,
$X, q^{(2n+1)}, Y \to X, p^{(2n+1)}, Y$, for $p \leq q$ w $(P, \leq)$.

Further, we consider derivations $X \Rightarrow Y$ in $F(P)$ (free pregroup generated by $(P, \leq)$). After Lambek Lambek (2001), we distinguish two special cases:

$(GCON)$ - generalized contraction:
$X, p^{(2n)}, q^{(2n+1)}, Y \to X, Y$;
$X, q^{(2n-1)}, p^{(2n)}, Y \to X, Y$; where $p \leq q$ in $(P, \leq)$.

$(GEXP)$ - generalized expansion:

$X, Y \to X, p^{(2n+1)}, q^{(2n)}, Y;$
$X, Y \to X, q^{(2n)}, p^{(2n-1)}, Y;$ where $p \leq q$ in $(P, \leq)$.

Relation $\Rightarrow$ is a reflexive and transitive closure of the relation $\to$.

Theorem 16 ***(Lambek switching lemma)***, *Lambek (1999)*
If $X \Rightarrow Y$ in $F(P)$, then there exist types U, V such that we can go from type X to U ($X \Rightarrow U$) only using generalized contractions, from type U to V ($U \Rightarrow V$) using only induced steps, and from type V to Y ($V \Rightarrow Y$) using only generalized expansions.

From the above mentioned lemma we get:

Corollary 17 *Buszkowski (2003) If $X \Rightarrow Y$ in $F(P)$, and Y is a simple type or an empty string, then X can be transformed into Y only by means of (CON) and (IND).*
If $X \Rightarrow Y$ in $F(P)$, and X is a simple type or an empty string, then X can be transformed into Y only by means of (EXP) and (IND).

7.2 Pregroups with modalities

In this section we generalize some definitions and results concerning pregroups introduced in Lambek (1999). The definition of a pregroup with β-operator was proposed by Fadda in Fadda (2002). The motivation to introduce modality operators was given by the fact there was a need to limit (locally) associativity in the calculus considered.

Definition 3 A pregroup with β-operator is a pregroup G enriched additionally with a monotone mapping $\beta : G \to G$.

Definition 4 β-pregroup is a pregroup with β-operator such that β-operator has the right adjoint $\hat{\beta}$ ($\hat{\beta}$- operator), ie. there exists a monotone mapping $\hat{\beta} : P \to P$ with the property that for all a and b in P, $\beta(a) \leq b$ if and only if $a \leq \hat{\beta}(b)$.

It is easy to show that $\hat{\beta}$-operators, if they exist, are uniquely defined and connected to β - operators with the following rules of expansion and contraction, for all $a \in P$.

$$a \leq \hat{\beta}(\beta(a)) \quad \text{and} \quad \beta(\hat{\beta}(a)) \leq a. \tag{7.7}$$

The basic rewriting rules are as follows:

1. Contracting rules

(CON) - contraction:

$X, p^{(n)}, p^{(n+1)}, Y \to X, Y;$

$(B - CON)$ - B-contraction:
$X, [B(Y)]^{(n)}, [B(Y)]^{(n+1)}, Z \to X, Z;$ where $B \in \{\beta, \hat{\beta}\}$.

$(\beta - CON)$ - β - contraction:
$X, [\beta(\hat{\beta}(Y))]^{(2n)}, Z \to X, Y^{(2n)}, Z;$
$X, [\hat{\beta}(\beta(Y))]^{(2n+1)}, Z \to X, Y^{(2n+1)}, Z;$

$(B - IND_c)$ - B_c induced step:
$X, [B(Y_1)]^{(2n)}, Z \to X, [B(Y_2)]^{(2n)}, Z;$
where $B \in \{\beta, \hat{\beta}\}$, and $Y_1 \to Y_2$ is a contracting rule.

$X, [B(Y_2)]^{(2n+1)}, Z \to X, [B(Y_1)]^{(2n+1)}, Z;$
where $B \in \{\beta, \hat{\beta}\}$, a $Y_1 \to Y_2$ is an expanding rule.

2. Expanding rules

(EXP) - expansion:
$X, Y \to X, p^{(n+1)}, p^{(n)}, Y;$

$(B - EXP)$ - B-expansion:
$X, Z \to X, [B(Y)]^{(n+1)}, [B(Y)]^{(n)}, Z;$ where $B \in \{\beta, \hat{\beta}\}$.

$(\beta - EXP)$ - β - expansion:
$X, Y^{(2n)}, Z \to X, [\hat{\beta}(\beta(Y))]^{(2n)}, Z;$
$X, Y^{(2n+1)}, Z \to X, [\beta(\hat{\beta}(Y))]^{(2n+1)}, Z.$

$(B - IND_e)$ - B_e induced step:
$X, [B(Y_1)]^{(2n)}, Z \to X, [B(Y_2)]^{(2n)}, Z;$
where $B \in \{\beta, \hat{\beta}\}$, a $Y_1 \to Y_2$ is an expanding rule.

$X, [B(Y_2)]^{(2n+1)}, Z \to X, [B(Y_1)]^{(2n+1)}, Z;$
where $B \in \{\beta, \hat{\beta}\}$, a $Y_1 \to Y_2$ is a contracting rule.

3. P-rules (neither expanding nor contracting)

(IND) - induced step:
$X, p^{(2n)}, Y \to X, q^{(2n)}, Y,$
$X, q^{(2n+1)}, Y \to X, p^{(2n+1)}, Y,$ for $p \leq q$ w $(P, \leq)$.

$(B - IND_p)$ - B_p induced step:

$X, [B(Y_1)]^{(2n)}, Z \to X, [B(Y_2)]^{(2n)}, Z;$
where $B \in \{\beta, \hat{\beta}\}$, and $Y_1 \to Y_2$ is a P-rule.

$X, [B(Y_2)]^{(2n+1)}, Z \to X, [B(Y_1)]^{(2n+1)}, Z;$
where $B \in \{\beta, \hat{\beta}\}$, and $Y_1 \to Y_2$ is a P-rule.

In above mentioned rules we assume that p, q are elements of P, whereas X, Y, Z, Y_1, Y_2 are elements of P'.

Relation $\Rightarrow$ is a reflexive and transitive closure of the relation $\to$.

In his work Fadda (2002) Fadda gives some examples illustrating the usage of β - pregroups for natural language. Among others, he shows that assigning a type $[\beta(X)]^r X [\beta(X)]^l$ to the conjunction *and* (where X is an arbitrary type), will let us see the structure of a sentence more clearly.
Consider the sentence: *John and Mary left.* Applying the calculus of pregroups without modalities we can show that the string of types assigned to given words can be reduced to the type of a sentence. However, the order of consecutive contraction is important here:
(np means a noun phrase).

(*)	**John**	**and**	**Mary**	**left.**		
	np	$np^r\ np\ np^l$	np	$np^r s$	$\to$	
		$np\ np^l$	np	$np^r s$	$\to$	
		np		$np^r s$	$\to$	s

(**)	**John**	**and**	**Mary**	**left.**		
	np	$np^r\ np\ np^l$	np	$np^r s$	$\to$	
		$np\ np^l$	np	$np^r s$	$\to$	
		$np\ np^l$		s	$\not\to$	s

In the second case (**) we do not get a type s. Applying the calculus of β-pregroups, we could handle the above mentioned sentence in the following way:

(**)	**John**	**and**	**Mary**	**left.**		
	$\beta(np)$	$[\beta(np)]^r np\ [\beta(np)]^l$	$\beta(np)$	$np^r s$	$\to$	s

In that case the structure of types 'induces' the order of contractions.

Normalization theorem for β - pregroups

Further we consider derivations of a type $X \Rightarrow Y$.

Definition 5 A derivation is called non-expanding, if there are no expanding rules present.

Definition 6 A derivation is called non-contracting, if there are no contracting rules present.

Definition 7 Composition of derivations $d_1(X \Rightarrow U)$ and $d_2(U \Rightarrow Y)$ is a derivation Y from X, which transforms first X into U according to d_1, and then U into Y according to d_2.

Definition 8 A derivation $d(X \Rightarrow Y)$ is called normal, if it is a composition of some non-expanding derivation $d_1(X \Rightarrow U)$ and some non-contracting derivation $d_2(U \Rightarrow Y)$.

On elements of P' we introduce a measure in the following way:

$\mu(\varepsilon) = 0,$
$\mu(p^{(n)}) = 1,$
$\mu(B(Y)) = \mu(Y) + 1, \quad \text{for } B \in \{\beta, \hat{\beta}\}$
$\mu(Y_1, ..., Y_k) = \mu(Y_1) + ... + \mu(Y_k).$

Measure on the rewriting rules is defined as follows:

$\mu(CON) = 2,$
$\mu(EXP) = 2,$
$\mu(\beta - CON) = 2,$
$\mu(\beta - EXP) = 2,$
$\mu(B - CON) = 2 + 2\mu(Y),$
$\mu(B - EXP) = 2 + 2\mu(Y),$
$\mu(IND) = 1,$
$\mu(B_c - IND) = 1 + \mu(d(Y_1 \to Y_2)),$
$\mu(B_e - IND) = 1 + \mu(d(Y_1 \to Y_2)),$
$\mu(B_p - IND) = 1 + \mu(d(Y_1 \to Y_2)),$

$\mu(d(X_0 \Rightarrow X_k)) = \mu(d(X_0 \to X_1)) + ... + \mu(d(X_{k-1} \to X_k)),$
where $X_0 \Rightarrow X_k$ means $X_0 \to X_1 \to ... \to X_k$.

Definition 9 A derivation $d(X \Rightarrow Y)$ is called minimal, if it has the least possible measure of all derivations Y from X, and the least possible complexity (which is understood as a sum of measures of all rules used in the derivation).

Definition 10 The position of a given rule in the derivation $X_0 \to X_1 \to ... \to X_n$ is number i, such that $X_{i-1} \to X_i$ is the occurrence of this rule in the derivation.

Definition 11 A degree of non-normal derivation $d(X \Rightarrow Y)$ is the minimal position of a contracting rule which occurs (not necessarily directly) after an expanding rule.
A degree of normal derivation is number 0.

Theorem 18 ***(Normalization theorem for β - pregroups).***
Every minimal derivation is normal.

Proof. Let $X_0 \to X_1 \to ... \to X_n$ be a minimal derivation. Let i be a degree of this derivation. We will show that $i = 0$, and as a consequence our derivation is normal.
Assume that $i > 0$. Of course $1 < i \leq n$ from the definition of a degree.
Let j be the greatest number less than i, such that $X_{j-1} \to X_j$ is the occurrence of an expanding rule.
Let R_1 denote the rule used on the position j, and R_2 the rule used in the position i.
There are following cases to be considered:
1.1. $R_1 = (EXP)$ $R_2 = (CON)$,
1.2. $R_1 = (EXP)$ $R_2 = (B - CON)$,
1.3. $R_1 = (EXP)$ $R_2 = (\beta - CON)$,
1.4. $R_1 = (EXP)$ $R_2 = (B - IND_c)$,
2.1. $R_1 = (B - EXP)$ $R_2 = (CON)$,
2.2. $R_1 = (B - EXP)$ $R_2 = (B - CON)$,
2.3. $R_1 = (B - EXP)$ $R_2 = (\beta - CON)$,
2.4. $R_1 = (B - EXP)$ $R_2 = (B - IND_c)$,
3.1. $R_1 = (\beta - EXP)$ $R_2 = (CON)$,
3.2. $R_1 = (\beta - EXP)$ $R_2 = (B - CON)$,
3.3. $R_1 = (\beta - EXP)$ $R_2 = (\beta - CON)$,
3.4. $R_1 = (\beta - EXP)$ $R_2 = (B - IND_c)$,
4.1. $R_1 = (B - IND_e)$ $R_2 = (CON)$,
4.2. $R_1 = (B - IND_e)$ $R_2 = (B - CON)$,
4.3. $R_1 = (B - IND_e)$ $R_2 = (\beta - CON)$,
4.4. $R_1 = (B - IND_e)$ $R_2 = (B - IND_c)$,

In the proof of this theorem the above mentioned cases are considered. In all cases we assume that the rule R_1 occurs on the position j, and the rule R_2 on the position i. All steps $X_j \to X_{j+1} \to ... \to X_{i-1}$ consist of application of non-expanding and non-contracting rules. These must be of the form of either (IND) or $(B_p - IND)$. None of this steps cannot be independent from $X_{i-1} \to X_i$, as otherwise we could do the last of independent steps after R_2, getting the derivation with the same measure but the lower degree. We can also assume that none of this steps is not independent from $X_{j-1} \to X_j$; otherwise it would transform our derivation performing the first step before R_1, increasing the number j, and changing neither i nor $\mu(d(X \Rightarrow Y))$.
If the rules R_1 and R_2 are adjacent (without intermediate P-rules), we change the order in case they are independent from each other (getting the derivation of smaller complexity); in case they are dependent from each other we show

that this part of derivation can be transformed using rules of smaller complexity - thus showing that the initial derivation was not normal.
Considering the above mentioned sixteen cases we show, that non-expanding rules must always precede the non-contracting ones. Otherwise our derivation would be not minimal, which would be a contradiction to our assumption. Thus every minimal derivation must be normal.
As the proof is long and technical, we show as an example only one of above mentioned sixteen cases:

Case 1.1. $R_1 = (EXP)$ $R_2 = (CON)$,
$X_{j-1} \to X_j$ is of the form $S, T \to S, p^{(n+1)}, p^{(n)}, T$;
$X_{i-1} \to X_i$ is of the form $U, q^{(n)}, q^{(n+1)}, V \to U, V$.
The derivation $X_{j-1} \to X_j \to ... \to X_{i-1} \to X_i$ could be as follows:

$S, p_0^{(2n)}, T \to S, p_0^{(2n)}, p_k^{(2n+1)}, p_k^{(2n)}, T \to S, p_0^{(2n)}, p_{k-1}^{(2n+1)}, p_k^{(2n)}, T \to ...$
$\to S, p_0^{(2n)}, p_0^{(2n+1)}, p_k^{(2n)}, T \to S, p_k^{(2n)}, T$, (assuming $p_0 \leq p_1 \leq ... \leq p_k$), its measure is $\mu(d(X_{j-1} \Rightarrow X_i)) = 2 + k + 2 = k + 4$.

The above mentioned derivation can be changed by the derivation:

$S, p_0^{(2n)}, T \to S, p_1^{(2n)}, T \to ...S, p_{k-1}^{(2n)}, T \to S, p_k^{(2n)}, T$, (assuming $p_0 \leq p_1 \leq ... \leq p_k$). The measure of a new derivation is $\mu(d(X_{j-1} \Rightarrow X_i)) = k$ (k times the rule (IND) was used).
We get contradiction, as the measure of the second derivation is smaller. We showed that the initial derivation was not normal.

□

Corollary 19 *If $X \Rightarrow Y$ in a free β-pregroup, and Y is a simple type or an empty string, then Y can be derived from X only by means of non-expanding rules.*
If $X \Rightarrow Y$ in a free β-pregroup, and X is a simple type or an empty string, then Y can be derived from X only by means of non-contracting rules.

7.3 Axiom system for pregroups with modalities

The rewriting system given in the previous section can also be presented as the calculus of sequents in a Gentzen style. Let $(P, \leq)$ be fixed. Atoms and types are defined as before. *Sequents* are of the form $X \Rightarrow Y$, where X, Y are types. Axiom and inference rules are as follows:

(Id) $\quad X \Rightarrow X,$

(LA) $\quad \dfrac{X, Y \Rightarrow Z}{X, p^{(n)}, p^{(n+1)}, Y \Rightarrow Z}$ $\qquad$ (RA) $\quad \dfrac{X \Rightarrow Y, Z}{X \Rightarrow Y, p^{(n+1)}, p^{(n)}, Z}$

(LIND) $\quad \dfrac{X, q^{(2n)}, Y \Rightarrow Z}{X, p^{(2n)}, Y \Rightarrow Z}$ $\qquad$ (RIND) $\quad \dfrac{X \Rightarrow Y, p^{(2n)}, Z}{X \Rightarrow Y, q^{(2n)}, Z}$

$\dfrac{X, p^{(2n+1)}, Y \Rightarrow Z}{X, q^{(2n+1)}, Y \Rightarrow Z}$ $\qquad$ $\dfrac{X \Rightarrow Y, q^{(2n+1)}, Z}{X \Rightarrow Y, p^{(2n+1)}, Z}$

In rules (LIND) and (RIND) we assume that $p \leq q$ in P. X, Y, Z are any arbitrary types, p, q are arbitrary elements of P, for $n \in Z$.

(BLA) $\quad \dfrac{X, T \Rightarrow Z}{X, [B(Y)]^{(n)}, [B(Y)]^{(n+1)}, T \Rightarrow Z}$

(BRA) $\quad \dfrac{X \Rightarrow T, Z}{X \Rightarrow T, [B(Y)]^{(n+1)}, [B(Y)]^{(n)}, Z}$

(β LA) $\quad \dfrac{X, Y^{(2n)}, T \Rightarrow Z}{X, [\beta(\hat{\beta}(Y))]^{(2n)}, T \Rightarrow Z}$ $\qquad$ (β RA) $\quad \dfrac{X \Rightarrow T, Y^{(2n)}, Z}{X \Rightarrow T, [\hat{\beta}(\beta(Y))]^{(2n)}, Z}$

$\dfrac{X, Y^{(2n+1)}, T \Rightarrow Z}{X, [\hat{\beta}(\beta(Y))]^{(2n+1)}, T \Rightarrow Z}$ $\qquad$ $\dfrac{X \Rightarrow T, Y^{(2n+1)}, Z}{X \Rightarrow T, [\beta(\hat{\beta}(Y))]^{(2n+1)}, Z}$

(BLIND) $\quad \dfrac{X, [B(Y_2)]^{(2n)}, Z \Rightarrow T}{X, [B(Y_1)]^{(2n)}, Z \Rightarrow T}$ $\qquad$ (BRIND) $\quad \dfrac{X \Rightarrow T, [B(Y_1)]^{(2n)}, Z}{X \Rightarrow T, [B(Y_2)]^{(2n)}, Z}$

$\dfrac{X, [B(Y_1)]^{(2n+1)}, Z \Rightarrow T}{X, [B(Y_2)]^{(2n+1)}, Z \Rightarrow T}$ $\qquad$ $\dfrac{X \Rightarrow T, [B(Y_2)]^{(2n+1)}, Z}{X \Rightarrow T, [B(Y_1)]^{(2n+1)}, Z}$

In rules (BLA), (BRA), (BLIND) and (BRIND), $B \in \{\beta, \hat{\beta}\}$. Additionally, in rules (BLIND) we assume that $Y_1 \to Y_2$ arises as a result of a non-expanding rule in an even case, and a non-contracting rules in an odd case, in a rewriting system from a former section. In rules (BRIND) we assume that $Y_1 \to Y_2$

arises as a result of non-contracting rule in an even case, and non-expanding rule in an odd case, in a rewriting system form a former section.

The cut rule is of the form:

$$\text{(CUT)} \quad \frac{X \Rightarrow Y,\ Y \Rightarrow Z}{X \Rightarrow Z}.$$

Let MS denote the system axiomatized by (Id), (LA), (RA), (LIND), (RIND), (BLA), (BRA), (β - LA), (β - RA), (BLIND) and (BRIND). Let MS' denote the system MS enriched additionally with a cut rule (CUT).

7.3.1 Cut elimination for the systems with modalities

We show that for above mentioned systems the following theorems hold:

Theorem 20 *For all types X, Y, $X \Rightarrow Y$ holds in the sense of a rewriting system if and only if $X \Rightarrow Y$ is provable in MS'.*

Proof. Assume $X \Rightarrow Y$ holds in the sense of the rewriting system.Then, there exist types $Z_0, ..., Z_n$, $n \geq 0$, such that $Z_0 = X$, $Z_n = Y$, and $Z_{i-1} \rightarrow Z_i$, $1 \leq i \leq n$. We show that $Z_{i-1} \Rightarrow Z_i$ is provable in MS', for $1 \leq i \leq n$.
(Here we show it only for a few chosen cases)

1. If $Z_{i-1} \rightarrow Z_i$ is the case of (CON), so it is of the form $X, p^{(n)}, p^{(n+1)}, Y \rightarrow X, Y$, we apply (LA) to axiom $X, Y \Rightarrow X, Y$. We get $\frac{X, Y \Rightarrow X, Y}{X, p^{(n)}, p^{(n+1)}, Y \Rightarrow X, Y}$.

7. If $Z_{i-1} \rightarrow Z_i$ is the case of (IND), so it is of the form:

7.1. $X, p^{(2n)}, Y \rightarrow X, q^{(2n)}, Y$,for $p \leq q$, we apply (LIND) to axiom $X, q^{(2n)}, Y \Rightarrow X, q^{(2n)}, Y$. We get $\frac{X, q^{(2n)}, Y \Rightarrow X, q^{(2n)}, Y}{X, p^{(2n)}, Y \Rightarrow X, q^{(2n)}, Y}$. We can also apply (RIND) to axiom $X, p^{(2n)}, Y \Rightarrow X, p^{(2n)}, Y$. We get then $\frac{X, p^{(2n)}, Y \Rightarrow X, p^{(2n)}, Y}{X, p^{(2n)}, Y \Rightarrow X, q^{(2n)}, Y}$.

7.2. $X, q^{(2n+1)}, Y \rightarrow X, p^{(2n+1)}, Y$, for $p \leq q$,we apply (LIND) to axiom $X, p^{(2n+1)}, Y \Rightarrow X, p^{(2n+1)}, Y$. We get:
$\frac{X, p^{(2n+1)}, Y \Rightarrow X, p^{(2n+1)}, Y}{X, q^{(2n+1)}, Y \Rightarrow X, p^{(2n+1)}, Y}$. We can also apply (RIND) to axiom $X, q^{(2n+1)}, Y \Rightarrow X, q^{(2n+1)}, Y$. We get then:
$\frac{X, q^{(2n+1)}, Y \Rightarrow X, q^{(2n+1)}, Y}{X, q^{(2n+1)}, Y \Rightarrow X, p^{(2n+1)}, Y}$.

So, if $n = 0$, then $X \Rightarrow Y$ is an axiom (Id), if $n > 0$, then $X \Rightarrow Y$ is provable in MS', using cut rule (CUT).

Assume, that $X \Rightarrow Y$ is provable MS. We show that $X \Rightarrow Y$ holds in the sense of the rewriting system.

If $X \Rightarrow Y$ jest (Id), then the claim is true.
For inference rules we show, that if the premise (premises) holds (hold) in the rewriting system, then the conclusion holds in this system. (Again, only a few chosen cases.)

1. For (LA), the antecedent of the conclusion can be transformed into the antecedent of the premise by (CON).
7. For (βLA)the antecedent of the conclusion can be transformed into the antecedent of the premise by (β-CON).
11. For (CUT),if the premises hold in the rewriting system, then the conclusion also holds in this system, since $\Rightarrow$ is transitive. □

Theorem 21 ***(Cut elimination theorem)***
For all types X, Y, $X \Rightarrow Y$ is provable in MS if and only if $X \Rightarrow Y$ is provable in MS'.

Proof. The 'only if' part is obvious. If for all types X, Y, $X \Rightarrow Y$ is provable in MS (without CUT), it is also provable in MS'.

Assume that $X \Rightarrow Y$ is provable in MS'.
By the theorem 20, $X \Rightarrow Y$ holds in the rewriting system. From the theorem 18 there exists such type U, that $X \Rightarrow U$ holds only by using non-expanding rules, whereas $U \Rightarrow Y$ holds only by using non-contracting rules.
Thus, there exist types $Z_0, ..., Z_m$, $(m \geq 0)$, such that $Z_0 = X$, $Z_m = U$ and for all $1 \leq i \leq m$, $Z_{i-1} \rightarrow Z_i$ is a result of non-expanding rules. We show that $Z_i \Rightarrow U$ is provable in MS, for all $0 \leq i \leq m$.
$Z_m \Rightarrow U$ is an axiom (Id). Assume that $Z_i \Rightarrow U$ is provable in MS, $i > 0$. If $Z_{i-1} \rightarrow Z_i$ is (CON), then $Z_{i-1} \Rightarrow U$ is a result of applying (LA) to $Z_i \Rightarrow U$.
If $Z_{i-1} \rightarrow Z_i$ is $(B - CON)$, then $Z_{i-1} \Rightarrow U$ is a result of applying (BLA) to $Z_i \Rightarrow U$.
If $Z_{i-1} \rightarrow Z_i$ is $(\beta - CON)$, then $Z_{i-1} \Rightarrow U$ is a result of applying (βLA) to $Z_i \Rightarrow U$.
If $Z_{i-1} \rightarrow Z_i$ is (IND), then $Z_{i-1} \Rightarrow U$ is a result of application (LIND) to $Z_i \Rightarrow U$.
If $Z_{i-1} \rightarrow Z_i$ is $(B - IND_c)$, then $Z_{i-1} \Rightarrow U$ is a result of applying (BLIND) to $Z_i \Rightarrow U$.
If $Z_{i-1} \rightarrow Z_i$ is $(B - IND_p)$, then $Z_{i-1} \Rightarrow U$ is a result of applying (BLIND) to $Z_i \Rightarrow U$.

Now, there exist types $V_0, ..., V_n$, $n \geq 0$, such that $V_0 = U$, $V_n = Y$, an for all $1 \leq i \leq n$, $V_{i-1} \to V_i$ is a result of applying a non-contracting rule.
We show that $X \Rightarrow V_i$ is provable in MS, for all $0 \leq i \leq n$.
$X \Rightarrow V_0$ is provable in MS from the first part of the proof.
Assume that $X \Rightarrow V_{i-1}$ is provable in MS, $1 \leq i$.
If $V_{i-1} \to V_i$ is (EXP), then $X \Rightarrow V_i$ is a result of applying (RA) to $X \Rightarrow V_{i-1}$.
If $V_{i-1} \to V_i$ is $(B - EXP)$, then $X \Rightarrow V_i$ is a result of applying (BRA) to $X \Rightarrow V_{i-1}$.
If $V_{i-1} \to V_i$ is $(\beta - EXP)$, then $X \Rightarrow V_i$ is a result of applying (βRA) to $X \Rightarrow V_{i-1}$.
If $V_{i-1} \to V_i$ is (IND), then $X \Rightarrow V_i$ is a result of applying (RIND) do $X \Rightarrow V_{i-1}$.
If $V_{i-1} \to V_i$ is $(B - IND_e)$, then $X \Rightarrow V_i$ is a result of applying (BRIND) to $X \Rightarrow V_{i-1}$.
If $V_{i-1} \to V_i$ is $(B - IND_p)$, then $X \Rightarrow V_i$ is a result of applying (BRIND) to $X \Rightarrow V_{i-1}$.

Thus, we showed that $X \Rightarrow Y$ is provable in MS. $\square$

7.4 Conclusion

In this paper we presented pregroups with modalities. First, we presented them in the form of a rewriting system, then we proposed the sequent system for them and finally showed the connections between those two presentations. Using those connections we were able to prove the cut elimination theorem.

References

Buszkowski, Wojciech. 2003. Sequent systems for compact bilinear logic. *Mathematical Logic Quarterly* 49:467–474.

Fadda, Mario. 2002. Towards flexible pregroup grammars. In *New Perspectives in Logic and Formal Linguistics*, pages 95–112. Roma: Bulzoni Editore.

Lambek, Joachim. 1999. Type grammars revisited. In *Logical Aspects of Computational Linguistics*, pages 1–27. Berlin: LNAI 1582, Springer.

Lambek, Joachim. 2001. Type grammars as pregroups. *Grammars* 4:21–39.

8

Simpler TAG Semantics through Synchronization

REBECCA NESSON AND STUART SHIEBER

Keywords SYNCHRONOUS TREE-ADJOINING GRAMMAR, STAG SEMANTICS, QUANTIFIER SCOPE, LONG-DISTANCE WH-MOVEMENT, RAISING VERBS, ATTITUDE VERBS, ADVERBS, RELATIVE CLAUSES, PREPOSITIONAL PHRASES.

Abstract

In recent years Laura Kallmeyer, Maribel Romero, and their collaborators have led research on TAG semantics through a series of papers refining a system of TAG semantics computation. Kallmeyer and Romero bring together the lessons of these attempts with a set of desirable properties that such a system should have. First, computation of the semantics of a sentence should rely only on the relationships expressed in the TAG derivation tree. Second, the generated semantics should compactly represent all valid interpretations of the input sentence, in particular with respect to quantifier scope. Third, the formalism should not, if possible, increase the expressivity of the TAG formalism. We revive the proposal of using synchronous TAG (STAG) to simultaneously generate syntactic and semantic representations for an input sentence. Although STAG meets the three requirements above, no serious attempt had previously been made to determine whether it can model the semantic constructions that have proved difficult for other approaches. In this paper we begin exploration of this question by proposing STAG analyses of many of the hard cases that have spurred the research in this area. We reframe the TAG semantics problem in the context of the STAG formalism and in the process present a simple, intuitive base for further exploration of TAG semantics. We provide analyses that demonstrate how STAG can handle quantifier scope, long-distance WH-movement, interaction of raising verbs and adverbs, attitude verbs and quantifiers, relative clauses, and quantifiers within prepositional phrases.

FG-2006.
Edited by Paola Monachesi, Gerald Penn,
Giorgio Satta and Shuly Wintner.

8.1 Introduction

In recent years Laura Kallmeyer, Maribel Romero, and their collaborators have led research on TAG semantics through a series of papers refining a system of TAG semantics computation using evolving techniques including enriched derivation tree structure (Kallmeyer, 2002a,b), flexible composition of feature-based TAG with a semantic representation associated with each elementary tree (Kallmeyer and Joshi, 2003, Joshi et al., 2003, Kallmeyer, 2003), semantic features in a more expressive extension of feature-based TAG (Gardent and Kallmeyer, 2003), and, most recently, semantic features on the derivation tree itself (Kallmeyer and Romero, 2004, Romero et al., 2004). Kallmeyer and Romero (2004) bring together the lessons of these attempts with a set of desirable properties that such a system should have. First, computation of the semantics of a sentence should rely only on the relationships expressed in the TAG derivation tree. Because TAG elementary trees represent minimal semantic units, the only information necessary for semantic computation should be the information encoded in the derivation tree: which elementary trees have combined and the address at which the combining operation took place. Second, the generated semantics should compactly represent all valid interpretations of the input sentence, in particular with respect to quantifier scope. Third, the formalism should not, if possible, increase the expressivity of the TAG formalism.

We revive the proposal of using synchronous TAG (STAG) to simultaneously generate syntactic and semantic representations for an input sentence (Shieber and Schabes, 1990). Although STAG meets the three requirements above, no serious attempt had previously been made to determine whether it can model the semantic constructions that have proved difficult for other approaches. In this paper we begin exploration of this question by proposing STAG analyses of many of the hard cases that have spurred the research in this area. We reframe the TAG semantics problem in the context of the STAG formalism and in the process present a simple, intuitive base for further exploration of TAG semantics.

After reviewing STAG in Section 8.2, we provide analyses in Sections 8.3.1 through 8.3.4 for sentences that exemplify several hard cases for TAG semantics that have been raised by Kallmeyer and others in recent papers: quantifier scope (as exemplified by sentences (12) and (16), presented below along with the desired semantic interpretations), long-distance WH-movement (13), interaction of raising verbs and adverbs, attitude verbs and quantifiers (14,15,16), relative clauses (17), and quantifiers within prepositional phrases (18) (Kallmeyer and Romero, 2004, Romero et al., 2004, Joshi et al., 2003, Kallmeyer, 2003, Kallmeyer and Joshi, 2003).[1]

[1] We notate curried two-place relations $P(x)(y)$ as $P(y,x)$ for readability.

(12) Everyone likes someone.
$every(x, person(x), some(z, person(z), like(x,z)))$
$some(z, person(z), every(x, person(x), like(x,z)))$

(13) Who does Bill think Paul said John likes?
$who(y, think(bill, say(paul, like(john, y))))$

(14) Bill thinks John apparently likes Mary.
$think(bill, apparently(like(john, mary)))$

(15) John sometimes likes everyone.
$every(x, person(x), sometimes(like(john, x)))$
$sometimes(every(x, person(x), like(john, x)))$

(16) Bill thinks everyone likes someone.
$think(bill, every(x, person(x), some(z, person(z), likes(x,z))))$
$think(bill, some(z, person(z), every(x, person(x), likes(x,z))))$

(17) A problem whose solution is difficult stumped Bill.

$$\begin{array}{l} a(x, \quad and(\quad problem(x), \\ \qquad\qquad\qquad the(y, and(solution(y), poss(x,y)), isDifficult(y))), \\ \qquad\quad stumped(bill, x)) \end{array}$$

(18) Two politicians spy on someone from every city.

$$\begin{array}{l} two(x, \quad politician(x), \\ \qquad\quad every(z, \quad city(z), \\ \qquad\qquad\qquad some(y, person(y) \wedge from(z,y), \\ \qquad\qquad\qquad spyOn(x,y)))) \\ every(z, \quad city(z), \\ \qquad\quad some(y, \quad person(y) \wedge from(z,y), \\ \qquad\qquad\qquad two(x, politcian(x), spyOn(x,y)))) \\ two(x, \quad politician(x), \\ \qquad\quad some(y, \quad every(z, city(z), person(y) \wedge from(z,y)) \\ \qquad\qquad\qquad spyOn(x,y))) \\ some(y, \quad every(z, city(z), person(y) \wedge from(z,y)) \\ \qquad\quad two(x, politician(x), spyOn(x,y))) \end{array}$$

8.2 Introduction to Synchronous TAG

A tree-adjoining grammar (TAG) consists of a set of elementary tree structures and two operations, substitution and adjunction, used to combine these structures. The elementary trees can be of arbitrary depth. Each internal node is labeled with a nonterminal symbol. Frontier nodes may be labeled with either terminal symbols or nonterminal symbols and one of the diacritics $\downarrow$ or $*$. Use of the diacritic $\downarrow$ on a frontier node indicates that it is a *substitution node*. The *substitution* operation occurs when an elementary tree rooted in the

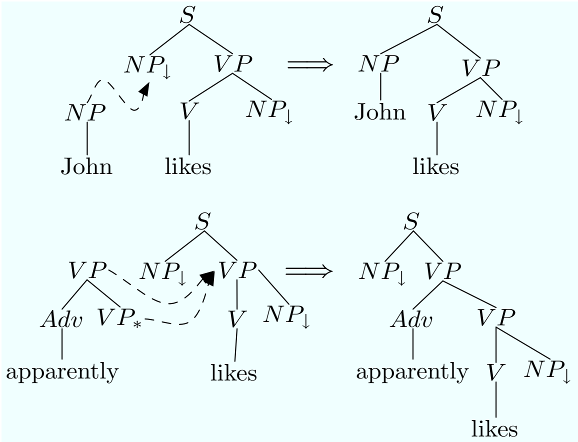


FIGURE 1 Example TAG substitution and adjuction operations.

nonterminal symbol *A* is substituted for a substitution node labeled with the nonterminal symbol *A*. Auxiliary trees are elementary trees in which the root and a frontier node, called the *foot node* and distinguished by the diacritic $*$, are labeled with the same nonterminal. The *adjunction* operation involves splicing an auxiliary tree with root and designated foot node labeled with a nonterminal *A* at a node in an elementary tree also labeled with nonterminal *A*. Examples of the substitution and adjunction operations on sample elementary trees are shown in Figure 1.

Synchronous TAG (STAG) extends TAG by taking the elementary structures to be pairs of TAG trees with links between particular nodes in those trees. An STAG is a set of triples, $\langle t_L, t_R, \frown \rangle$ where t_L and t_R are elementary TAG trees and $\frown$ is a linking relation between nodes in t_L and nodes in t_R (Shieber, 1994, Shieber and Schabes, 1990). Derivation proceeds as in TAG except that all operations must be paired. That is, a tree can only be substituted or adjoined at a node if its pair is simultaneously substituted or adjoined at a linked node. We notate the links by using boxed indices $\boxed{i}$ marking linked nodes.

Figure 2 contains a sample English syntax/semantics grammar fragment that can be used to parse the sentence "John apparently likes Mary". The node labels we use in the semantics correspond to the semantic types of the phrases they dominate.[2] Variables such as *x* in the semantic tree in Figure 3 are taken to be bound in the obvious way, so that in multiple uses of the tree

[2]This representation is for the sake of readability. The labels could be replaced using any well-chosen finite set of nonterminal symbols.

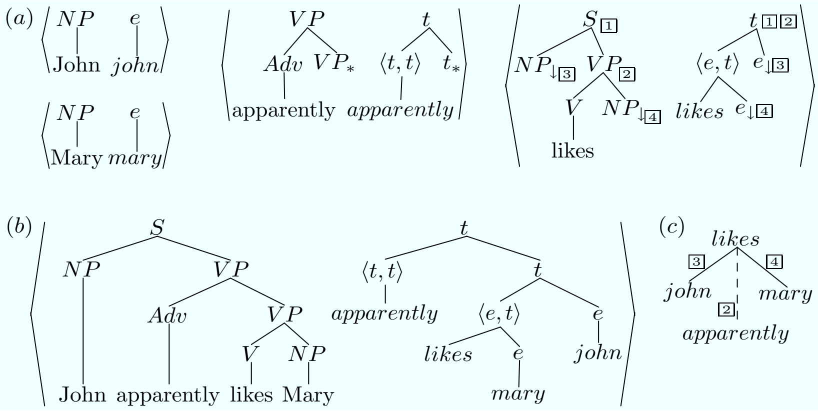


FIGURE 2 An English syntax/semantics STAG fragment (a), derived tree pair (b), and derivation tree (c) for the sentence “John apparently likes Mary.”

they can be presumed to be renamed apart.

Figure 2(c) shows the derivation tree for the sentence. Substitutions are notated with a solid line and adjunctions are notated with a dashed line. Note that each link in the derivation tree specifies a link number in the elementary tree pair. The links provide the location of the operations in the syntax tree and in the semantics tree. These operations must occur at linked nodes in the target elementary tree pair. In this case, the noun phrases *John* and *Mary* substitute into *likes* at links $\boxed{3}$ and $\boxed{4}$ respectively. The word *apparently* adjoins at link $\boxed{2}$. The resulting semantic representation can be read off the derived tree by treating the leftmost child of a node as a functor and its siblings as its arguments. Our sample sentence thus results in the semantic representation *apparently*(*likes*(*john*, *mary*)).

8.3 STAG Analyses of the Phenomena

8.3.1 Quantifier Scope and Wh-Words

For sentence (12), we would like to generate a scope-neutral semantic representation that allows both the reading where *some* takes scope over *every* and the reading where *every* takes scope over *some*. We propose a solution in which a derivation tree with multiple adjunction nondeterministically determines multiple derived trees each manifesting explicit scope (Schabes and Shieber, 1993); the derivation tree *itself* is therefore the scope neutral representation.

The multi-component quantifier approach followed by Joshi et al. (2003)

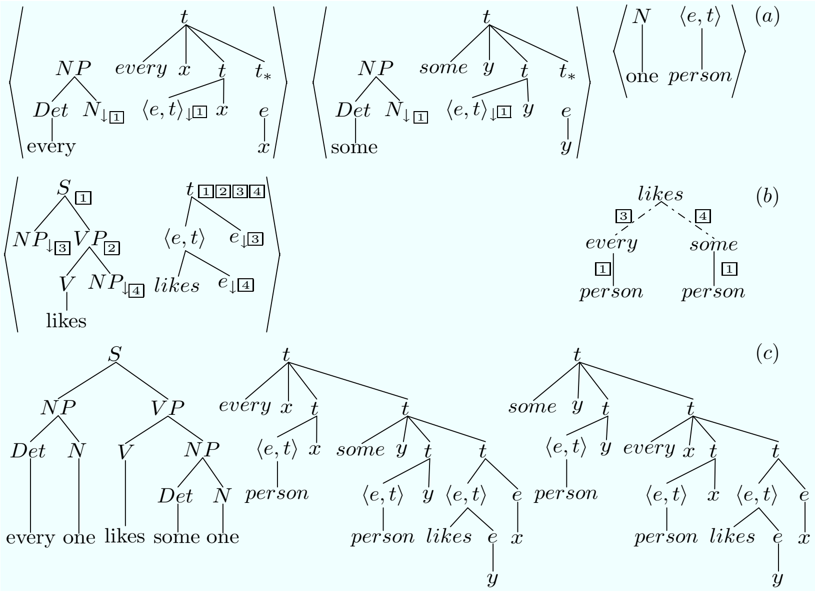


FIGURE 3 The elementary tree pairs (a), derivation tree (b), and derived syntactic and semantic trees (c) for the sentence "Everyone likes someone". Note that the derivation tree is a scope neutral representation: depending on whether *every* or *some* adjoins higher, different semantic derived trees and scope orderings are obtained.

suggests a natural implementation of quantifiers in STAG.[3] In this approach the syntactic tree for quantifiers has two parts, one that corresponds to the scope of the quantifier and attaches at the point where the quantifier takes scope, and the other that contains the quantifier itself and its restriction and attaches where syntactically expected at a noun phrase. In their work, a single-node auxiliary tree is used for the scope part of the syntax in order to get the desired relationship between the quantifier and the quantified expression in features threaded through the derivation tree and hence in the semantics. Using STAG, we do not need the single-node auxiliary tree in the syntax because we can pair the usual syntactic representation for quantified NPs with a multi-component semantic representation that expresses the same idea (Figure 3). In order to use these quantifiers, we change the links in the elementary trees for verbs to allow a single link to indicate two positions in the semantics

[3]The multi-component approach to quantifiers in STAG was first suggested by Shieber and Schabes (1990) under the rewriting definition of STAG derivation where the order of rewriting produced the scope ambiguity. Williford (1993) explored the use of multiple adjunction to achieve scope ambiguity.

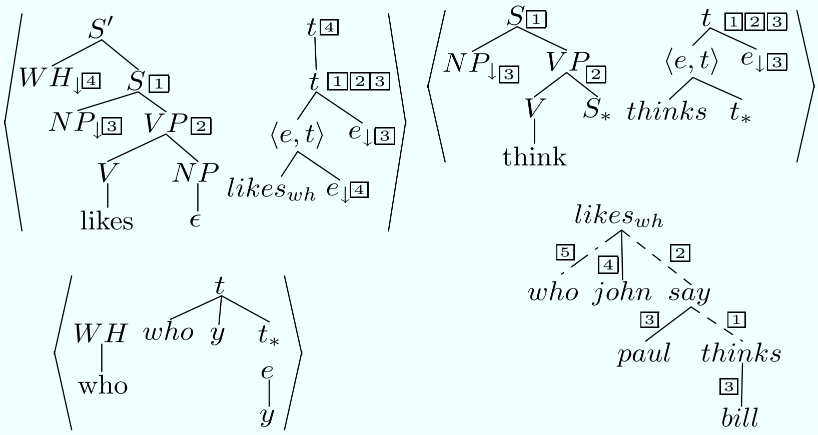


FIGURE 4 Selection of elementary trees and full derivation tree for the sentence "Who does Bill think Paul said John likes?".

where a tree pair can adjoin, as shown in Figure 3.[4]

Given this representation of quantifiers we get the derivation tree shown in Figure 3 for sentence (12).[5] Note that the resulting derivation tree necessarily incorporates *multiple adjunction* (Schabes and Shieber, 1993), that is, multiple auxiliary trees are adjoined at the same node in an auxiliary tree. In particular, the scope parts of both *every* and *some* attach at the root of the semantic tree of *likes*. Such cases of multiple adjunction induce ambiguity; the derivation tree represents multiple derived trees. In the case at hand, the derivation is ambiguous as to which quantifier scopes higher than the other. This ambiguity in the derivation tree thus models the set of valid scopings for the sentence. In essence, this method uses multiple adjunction to model scope-neutrality.

This same method can be used to obtain the correct scope relations for sentences with long-distance WH-movement such as sentence (13) using the multi-component elementary tree pair for *who* and the elementary tree pairs for *thinks* (the tree pair for *says* is similar) and *likes* in the WH context given in Figure 4. Kallmeyer and Romero (2004) highlight this case as difficult because in the usual syntactic analysis there is no link in the derivation tree

---

[4]We have chosen here to add the three-way links in addition to the existing links in the tree for unquantified noun phrases such as proper nouns (though we suppress the two-way NP links in the figures for readability). Another possibility would be to remove the two-way links. In this case, all noun phrases would be "lifted" à la Montague. That is, even unquantified noun phrases would have a scope part, which could be a single-node auxiliary tree.

[5]We notate multi-component insertions that involve both a substitution and an adjunction with a combination dashed and dotted line.

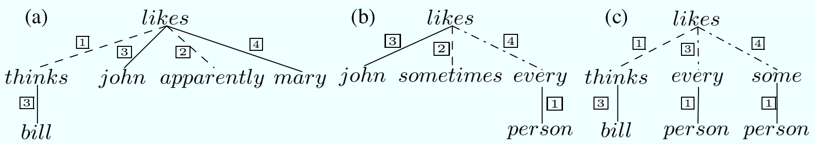


FIGURE 5 Derivation trees for (a) “Bill thinks John apparently likes Mary”, (b) “John sometimes likes everyone”, and (c) “Bill thinks everyone likes someone.”

between *who* and *thinks* or between *thinks* and *likes*, but in the desired semantics *who* takes scope over the *thinks* proposition and the *likes* proposition is an argument to *thinks*.

In our analysis, by contrast, the semantics follows quite naturally from the standard syntactic analysis of the structure of the *likes* elementary tree in the WH context and the elementary tree pair for *thinks* given in Figure 4. The derivation of this sentence is also given in Figure 4. Note that it is required by the structure of the trees that *who* take scope over *thinks*.

8.3.2 The Interaction Between Attitude Verbs, Raising Verbs, Adverbs and Quantifiers

The interaction between attitude verbs and raising verbs or adverbs as in sentences (14), (15), and (16) has been problematic for TAG semantics (Kallmeyer and Romero, 2004). A successful analysis must be flexible enough to produce the correct semantics for sentence (14) even though there is no link between *thinks* and *apparently* in the derivation tree. It must also be flexible enough to allow all scope orderings between VP modifiers and quantifiers as in sentence (15). In fact, given the elementary trees we have already presented and the ones for attitude verbs demonstrated by Figure 4, our analysis already allows for scope interactions among all these elements. Indeed, because the semantic components of attitude verbs, VP modifiers, and quantifiers all adjoin at the same node in the semantic tree of the verb, our analysis allows all scope orderings among them. This is clearly too permissive, because it allows quantifiers to scope out of the finite clause in which they appear and would allow a reading of sentence (14) in which *apparently* scopes over *thinks*. To prevent quantifiers from scoping out of the finite clause in which they appear, as in sentences (14) and (16), we can add an additional adjunction site to the semantic trees for verbs above the current root node. This is shown in Figure 6 in the $likes_2$ tree pair. The link configuration ensures that attitude verbs (adjoining at link $\boxed{1}$) will now scope higher than all VP modifiers (adjoining at $\boxed{2}$) and quantifiers (adjoining at links $\boxed{3}$ and $\boxed{4}$). VP modifiers and quantifiers will still be able to take all scope orderings relative to each other. Using the modified verb trees, STAG produces the correct semantics for sentences (14),

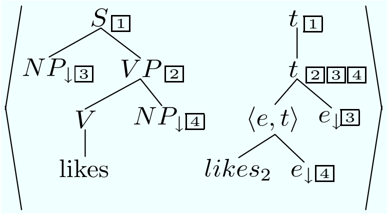


FIGURE 6 Modified tree for *likes* that enforces a restriction on quantifiers scoping outside of the finite clause.

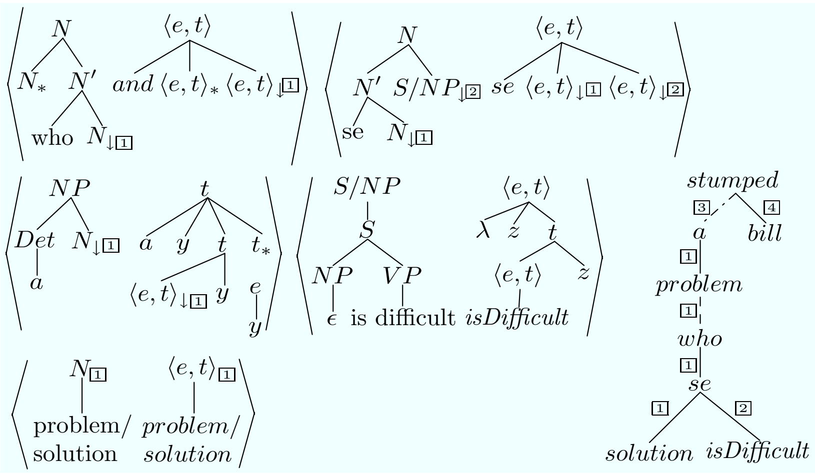


FIGURE 7 Key elementary trees and derivation for "A problem whose solution is difficult stumped Bill."

(15), and (16) with the derivations given in Figure 5.

8.3.3 Relative Clauses

Relative clauses provide another putatively difficult case for TAG semantics because both the main verb and the relative clause need access to the variable introduced by the determiner as in sentence (17) (Kallmeyer, 2003). We overcome this difficulty and compute the desired semantics by introducing higher-order functions into the semantic trees using lambda-calculus notation. This modification allows us to maintain tree-locality. The syntactic analysis we use is similar to that of Kallmeyer (2003) in that it maintains the *Condition on Elementary Tree Minimality* (Frank, 1992) and uses the relative pronoun to introduce the relative clause. However, it treats the relative pronoun as a noun modifier rather than a noun phrase modifier.

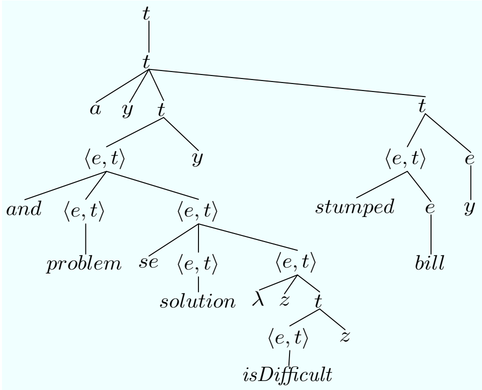


FIGURE 8 Derived tree for "A problem whose solution is difficult stumped Bill."

We also posit the existence of "lifted" versions of the elementary trees for verbs in which their argument positions have been abstracted over. We use a higher-order conjunction *and* that relates two properties: $\lambda PQx.P(x) \wedge Q(x)$, and a higher-order *se* function that relates two properties and makes use of the higher-order conjunction: $\lambda PQx.the(y, and(P, \lambda z.poss(x,z))(y), Q(y))$. The elementary tree pairs and resulting derivation tree for sentence (17) are given in Figure 7. The derived tree is given in Figure 8. When reduced, the resulting semantics is $a(z, \lambda x.(problem(x) \wedge the(y, solution(y) \wedge poss(x,y), isDifficult(y))), stumped(bill, z))$.

8.3.4 Nested Quantifiers and Inverse Linking

Quantifiers in prepositional phrases such as in sentence (18) pose another challenge for TAG semantics (Joshi et al., 2003). Although a nested quantifier may take scope over the quantifier within which it is nested (so-called "inverse linking") not all permutations of scope orderings of the quantifiers are available (Joshi et al., 2003). In particular, readings in which a quantifier intervenes between a nesting quantifier and its nested quantifier are not valid. In our example sentence (18), this predicts that the readings *some* > *two* > *every* and *every* > *two* > *some* should not be valid. Joshi et al. (2003) introduce a special device allowing nesting and nested quantifiers to form an indivisible quantifier set during the derivation, which prevents other quantifiers from intervening between them. In our solution, because the nested quantifier is introduced through the prepositional phrase, which in turn modifies the noun phrase containing the nesting quantifier, the two quantifiers already naturally

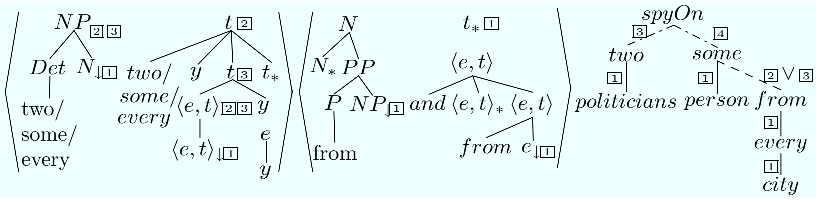


FIGURE 9 Key elementary trees and derivations for "Two politicians spy on someone from every city."

form a set that operates as a unit with respect to the rest of the derivation.[6] The elementary tree pairs and derivation trees for our analysis of (18) are shown in Figure 9.

One notable feature of this analysis is that the four different scope readings that result are not the product of a single derivation tree. The alternate scope orderings for the nested and nesting quantifier exist because there are two available adjunction sites for the scope of quantifiers in the prepositional phrase to attach. This results in two distinct derivation trees. The alternate scope orderings for this quantifier set and the remaining quantifier are obtained by multiple adjunction at the root of the verb tree. The set of valid derivation trees for a sentence thus constitutes the scope neutral representation. This set of trees may be compactly represented, for instance as a shared forest.[7]

8.4 Comparison to the Kallmeyer and Romero Approach

As mentioned above, research on TAG semantics has been led by Laura Kallmeyer, Maribel Romero, and their collaborators through a series of papers refining a system of TAG semantics computation using feature unification and other formal devices (Kallmeyer and Romero, 2004, Romero et al., 2004, Kallmeyer, 2003, Kallmeyer and Joshi, 2003, Joshi et al., 2003, Gardent and Kallmeyer, 2003). Although their approach has evolved over time,

[6]We make use of tree-set-local TAG in the semantics where the tree set for *every* adjoins into the tree set for $from$. Although tree-set-local TAG is more powerful than TAG, this particular use is benign because it cannot be iterated. More concretely, we could conventionally make the grammar tree-local by including all combinations of prepositions with quantifiers as elementary trees in the grammar.

[7]This analysis, like that of Joshi et al. (2003), makes several predictions about quantifier scope that might be disputed. First, some argue that more than four scope orderings should be available for sentences like sentence (18) (VanLehn, 1978, Hobbs and Shieber, 1987). This analysis cannot generate additional scope orderings without breaking tree set locality. Second, the scope readings in which the nesting quantifier takes scope over the nested quantifier result in the nested quantifier having scope over the restriction of the nesting quantifier but not over its scope. Donkey sentence constructions such as "Every man with two books loves them" call this prediction into question.

the underlying principles of using the relationships expressed in the derivation tree as the basis for the computation and generating underspecified semantic representations have been constant. In its current formulation, they perform semantic computation by attaching semantic feature structures directly to the nodes in the derivation tree. When carefully chosen, these features unify to produce an underspecified representation of the semantics of a sentence that, when further disambiguated, generates the set of valid interpretations. In one or another of their recent papers they have provided successful analyses of each of the hard cases that we have addressed here, though some of their analyses might have to be restated to bring them up to date with the newest formulation of their method.

Our work owes much to theirs both for the clear formulation of the problems and the progress in formulating analyses for some of the hard cases. The primary advantage of our approach is its conceptual simplicity. The clear separation of syntax and semantics, the directness of the link interface, and the familiarity of the TAG operations used in our approach make it very simple. The semantic-feature-unification-based approach has become cleaner and easier to understand as Kallmeyer and others have refined it over the years. Nonetheless, it is safe to say that the amount of formal machinery—including propositional labels, separate individual and propositional variables, semantic representations consisting of a set of formulas and a set of scope constraints, features on the derived tree and the derivation tree, each semantic feature structure containing a nested feature structure for each address in the elementary syntax tree, each of these feature structures containing features to handle binding of propositional and individual variables, feature unification, flexible composition, and quantifier sets—necessary to solve the range of problems that we have addressed here, is qualitatively more complex. In fact, we use no formal machinery that had not been introduced by 1994 in the TAG literature.

An additional advantage of our approach is that it does not increase the expressivity of the TAG formalism. One might think that the inclusion of multiple adjunction would lead to an increase in expressivity (Dras, 1999). However, because links can only be used once in an STAG derivation, only a finite number of multiple adjunctions may occur at a single adjunction site. This rules out problematic uses of multiple adjunction. Kallmeyer and Romero maintain the semantic features on the derivation tree rather than in the feature structures already used in the feature-based TAGs (FTAG) of their syntax in part because the set of semantic feature structures is not finite, potentially increasing the expressivity of the FTAG formalism (Kallmeyer and Romero, 2004). Although moving the features to the derivation tree avoids increasing the expressivity of the formalism used for syntax when taken alone, the additional expressivity in the features of the semantics could be used to block operations in the syntax thereby filtering the syntax to produce non-tree-adjoining languages. It remains to be seen whether this additional expressivity will be required for TAG semantics.

Advantages and disadvantages of the different methods aside, in this still nascent area of research it is desirable to have several quite different approaches at our disposal as we explore the hard problems presented by generating natural language semantics in the TAG framework. Our approach revives an old idea with the aim of opening a new avenue for research into semantics in the TAG framework.

8.5 Conclusion

We have presented the synchronous TAG formalism as a method for computing semantics in the TAG framework, and have shown that it enables simple, natural analyses for all of the cases that have exercised recent attempts at formulating formal semantics for TAG. It satisfies each of the desiderata laid out at the beginning of this paper. First, it does not require any additional information other than that available in the derivation tree to generate the semantics. Because the syntax and semantic representations are built up synchronously, the derivation tree set is a complete specification of the relationship between them. Nothing other than the set of elementary tree pairs and the synchronous TAG operations are required to generate a semantic representation. Second, the derivation tree set provides a compact representation for all valid semantic interpretations of the given sentence. Using multiply-adjoined quantifiers we take advantage of the ambiguity in the interpretation of the derivation tree that is introduced by multiple adjunction. We take each possible ordering of multiply-adjoined trees to be valid. We leave open the possibility of using an additional method to prefer certain scope orders and disprefer or eliminate others. Third, the STAG system, as used, does not increase the expressivity of the TAG formalism (Shieber, 1994). Finally, our analysis is a straightforward expression of a simple idea: we use TAG for both syntax and semantics and use the derivation tree and the links between trees in elementary tree pairs as the interface between them.

8.6 Acknowledgements

This work was supported in part by grant IIS-0329089 from the National Science Foundation. We wish to thank Rani Nelken and the three anonymous reviewers for valuable comments on earlier drafts.

References

Dras, Mark. 1999. A meta-level grammar: Redefining synchronous TAG for translation and paraphrase. In *Proceedings of the Thirty-Seventh Annual Meeting of the*

Association for Computational Linguistics, pages 80–87. Maryland, USA.

Frank, Robert. 1992. Syntactic locality and Tree Adjoining Grammar: Grammatical, acquisition and processing perspectives. Ph.D. Thesis, University of Pennsylvania.

Gardent, Claire and Laura Kallmeyer. 2003. Semantic construction in feature-based TAG. In *Proceedings of the 10th Meeting of the European Chapter of the Association for Computational Linguistics*. Budapest, Hungary.

Hobbs, Jerry and Stuart M. Shieber. 1987. An algorithm for generating quantifier scopings. *Computational Linguistics* 13(1-2):47–63.

Joshi, Aravind K., Laura Kallmeyer, and Maribel Romero. 2003. Flexible composition in LTAG: Quantifier scope and inverse linking. In I. v. d. S. Harry Bunt and R. Morante, eds., *Proceedings of the Fifth International Workshop on Computational Semantics IWCS-5*, pages 179–194. Tilburg.

Kallmeyer, Laura. 2002a. Enriching the TAG derivation tree for semantics. In S. Busemann, ed., *KONVENS 2002. 6. Konferenz zur Verarbeitung natürlicher Sprache.*, pages 67–74. Saarbrücken.

Kallmeyer, Laura. 2002b. Using an enriched tag derivation structure as basis for semantics. In *Proceedings of the Sixth International Workshop on Tree Adjoining Grammar and Related Frameworks (TAG+6)*, pages 127–136. Venice.

Kallmeyer, Laura. 2003. LTAG semantics for relative clauses. In I. v. d. S. Harry Bunt and R. Morante, eds., *Proceedings of the Fifth International Workshop on Computational Semantics IWCS-5*, pages 195–210. Tilburg.

Kallmeyer, Laura and Aravind K. Joshi. 2003. Factoring predicate argument and scope semantics: Underspecified semantics with LTAG. *Research on Language and Computation* 1:3–58.

Kallmeyer, Laura and Maribel Romero. 2004. LTAG semantics with semantic unification. In *Proceedings of TAG+7*, pages 155–162. Vancouver.

Romero, Maribel, Laura Kallmeyer, and Olga Babko-Malaya. 2004. LTAG semantics for questions. In *Proceedings of TAG+7*, pages 186–193. Vancouver.

Schabes, Yves and Stuart M. Shieber. 1993. An alternative conception of tree-adjoining derivation. *Computational Linguistics* 20(1):91–124.

Shieber, Stuart M. 1994. Restricting the weak-generative capacity of synchronous tree-adjoining grammars. *Computational Intelligence* 10(4):371–385.

Shieber, Stuart M. and Yves Schabes. 1990. Synchronous tree-adjoining grammars. In *Proceedings of the 13th International Conference on Computational Linguistics*, vol. 3, pages 253–258. Helsinki.

VanLehn, Kurt. 1978. Determining the scope of English quantifiers. Tech. Rep. 483, MIT Artificial Intelligence Laboratory, Cambridge, MA.

Williford, Sean. 1993. Application of synchronous tree-adjoining grammar to quantifier scoping phenomena in English. Undergraduate Thesis, Harvard College.

9

Encoding second order string ACG with Deterministic Tree Walking Transducers.

Sylvain Salvati

9.1 Introduction

Abstract Categorial Grammars (ACGs) (de Groote (2001)) are based on the linear logic (Girard (1987)) and on the linear λ-calculus. They describe the surface structures by using for syntax the ideas Montague (1974) devoted to semantics. ACGs describe parse structures with higher-order linear λ-terms and syntax as a higher-order linear homomorphism (lexicon) on parse structures. Intuitively, the higher the order of the parse structures is, the richer should the languages of analysis be and the higher the order of the lexicons is, the richer should the class of languages be. On the one hand, de Groote and Pogodalla (2004) have shown how to encode of several context free formalisms by using second order parse structures (*i.e.* regular sets of trees). They have encoded Context Free Grammars using second order lexicons, Linear Context Free Tree Grammars using third order lexicons and Linear Context Free Rewriting Systems (Weir (1988)) with fourth order lexicons. On the other hand Yoshinaka and Kanazawa (2005) have explored the expressivity of lexicalized ACGs. They have exhibited a non-semilinear string language with third order parse structures and an NP-complete string language with fourth order parse structures. (Salvati (2005) gave an example of an NP-complete language with third order parse structures and a first order lexicon).

The present work addresses the problem of the expressivity of ACGs in a particular case. We show that the class of languages defined by second order string ACGs is the same as the class of languages defined as outputs of

FG-2006.
Edited by Paola Monachesi, Gerald Penn,
Giorgio Satta and Shuly Wintner.

Deterministic Tree Walking Transducers (DTWT) (Aho and Ullman (1971)). Together with the results of de Groote and Pogodalla (2004) and Weir (1992), this result proves that the generative power of second order string ACGs is exactly the same as the generative power of Linear Context Free Rewriting Systems. This furthermore shows that second order string ACGs can always be described with fourth order lexicons. We may nevertheless conjecture that the use of lexicons of order greater than four may give more compact grammars.

The paper is organized as follows: we first briefly define the linear λ-calculus and ACGs in section 9.2. In section 9.3, we use the correspondence between proofs of linear logic and linear λ-terms to relate subformulae of a type α with subterms of terms of type α. Section 9.4 introduces *h*-reduction, the reduction used by the DTWTs which encode second order string ACGs. Section 9.5 presents the encoding of second order string ACGs with DTWTs. Finally we conclude and outline future work in section 9.6.

9.2 Definitions

Given a finite set of atomic types $\mathcal{A}$, we define, $\mathcal{T}_{\mathcal{A}}$, the set of linear applicative types built on $\mathcal{A}$ with the following grammar:

$$\mathcal{T}_{\mathcal{A}} ::= \mathcal{A} \mid (\mathcal{T}_{\mathcal{A}} \multimap \mathcal{T}_{\mathcal{A}})$$

If $\alpha_1, \ldots, \alpha_n$ are elements of $\mathcal{T}_{\mathcal{A}}$ and $\alpha \in \mathcal{A}$ we will write $(\alpha_1, \ldots, \alpha_n) \multimap \alpha$ the type $(\alpha_1 \multimap (\cdots(\alpha_n \multimap \alpha)\cdots))$. The order of the type α, $\mathrm{ord}(\alpha)$, is 1 if α is atomic (*i.e.* $\alpha \in \mathcal{A}$), and $\mathrm{ord}(\alpha \multimap \beta) = \max(\mathrm{ord}(\alpha) + 1, \mathrm{ord}(\beta))$.

Higher-order signatures are triples $(C, \mathcal{A}, \tau)$ where C is a finite set of constants, $\mathcal{A}$ is a finite set of atomic types and τ is a function from C to $\mathcal{T}_{\mathcal{A}}$. The order of a signature $(C, \mathcal{A}, \tau)$ is $\max\{\mathrm{ord}(\tau(a)) | a \in C\}$. Given a higher-order signature $\Sigma = (C, \mathcal{A}, \tau)$ we will denote $\mathcal{A}$ by $\mathcal{A}_\Sigma$, C by C_Σ, τ by τ_Σ and $\mathcal{T}_{\mathcal{A}}$ by $\mathcal{T}_\Sigma$; if $\tau_\Sigma(a) = (\alpha_1, \ldots, \alpha_n) \multimap \alpha$, then the arity of $a \in C_\Sigma$ is n, it will be noted ρ_a^Σ or ρ_a (when Σ is clear from the context).

A higher-order signature Σ is said to be a *string signature* if $\mathcal{A}_\Sigma = \{*\}$, $\# \in C_\Sigma$, $\tau_\Sigma(\#) = *$ and for all $a \in C_\Sigma \backslash \{\#\}$, $\tau_\Sigma(a) = (* \multimap *)$.

We are now going to define the set of linear λ-terms built on a signature Σ. We assume that the notions of free variables[1], capture-avoiding substitutions, α-conversion, β-reduction, η-reduction… are familiar to the reader. If necessary, one may consult Barendregt (1984).

Given a higher-order signature Σ and $\alpha \in \mathcal{T}_\Sigma$, we assume that we are given an infinite enumerable set of variables x^α, y^α, $z^\alpha \ldots$, Λ_Σ^α the set of linear λ-terms of type α built on Σ is the smallest set verifying:

1. if $a \in C_\Sigma$ and $\tau_\Sigma(a) = \alpha$ then $a \in \Lambda_\Sigma^\alpha$
2. $x^\alpha \in \Lambda_\Sigma^\alpha$

[1]Given a λ-term t, we will write $FV(t)$ to denote the set of its free variables.

3. if $t_1 \in \Lambda_\Sigma^{(\beta\multimap\alpha)}$, $t_2 \in \Lambda_\Sigma^{\beta}$ and $FV(t_1) \cap FV(t_2) = \emptyset$ then $(t_1 t_2) \in \Lambda_\Sigma^{\alpha}$
4. if $t \in \Lambda_\Sigma^{\beta}$, $x^\alpha \in FV(t)$ then $\lambda x^\alpha.t \in \Lambda_\Sigma^{(\alpha\multimap\beta)}$

The set Λ_Σ denotes $\bigcup_{\alpha\in\mathcal{T}_\Sigma} \Lambda_\Sigma^\alpha$. Linear λ-terms are *linear* because variables may occur free at most once in them and that whenever $\lambda x^\alpha.t$ is a linear λ-term, x^α has exactly one free occurrence in t. Moreover, whenever $t \in \Lambda_\Sigma^\alpha \cap \Lambda_\Sigma^\beta$ then $\alpha = \beta$, *i.e.* every linear λ-term has a unique type in a given signature Σ.

We may, when it is not relevant, strip the typing annotation from the variables. We will write $\lambda x_1 \ldots x_n.t$ for the term $\lambda x_1. \ldots \lambda x_n.t$ and $t_0 t_1 \ldots t_n$ for $(\ldots(t_0 t_1)\ldots t_n)$. Given a list of indices $S = [i_1, \ldots, i_n]$, we will write $\lambda\overrightarrow{x_S}.t$ the term $\lambda x_{i_1} \ldots x_{i_n}.t$, $t_0\overrightarrow{t_S}$ the term $t_0 t_{i_1} \ldots t_{i_n}$ and $\overrightarrow{c_S}t$ the term $c_{i_1}(\ldots c_{i_n}(t)\ldots)$ when for all $j \in [1, n]$, c_{i_j} has type $* \multimap *$. In particular, $\lambda\overrightarrow{x_n}.t$, $t_0\overrightarrow{t_n}$ and $\overrightarrow{c_n}t$ may be used when $S = [1, \ldots, n]$.

Given a string signature Σ, strings will be represented by the closed terms of type $*$. For example, the term $c_1(\ldots(c_n\#)\ldots)$ represents the string $c_1 \ldots c_n$; given w, a string built on C_Σ, $/w/$ will denote the term of Λ_Σ^* which is in normal form and represents w.

To define the subterms of $t \in \Lambda_\Sigma$, we follow Huet (1997) and consider them as pairs $(C[], t')$ (where $C[]$ is a context, *i.e.* a term with a hole) such that $t = C[t']$. The set of subterms of t is denoted by $\mathcal{S}_t$. In particular, we define $\mathcal{S}_t^\alpha$ to be $\{(C[], v) \in \mathcal{S}_t | v \in \Lambda_\Sigma^\alpha\}$. If x is free in t, we note $C_{t,x}[]$ the context such that $C_{t,x}[x] = t$ and x is not free in $C_{t,x}[]$. Remark that since t is linear $C_{t,x}[]$ is uniquely defined.

We say that a term t is in long from if for all $(C[], t') \in \mathcal{S}_t^{\alpha\multimap\beta}$ either $t' = \lambda x.t''$ or $C[] = C'[[]t'']$. Every term can be put in long form by η-expansion, therefore if t is the long form of t', then $t \xrightarrow{*}_\eta t'$. When a term is in long form, all its possible arguments are abstracted by a λ-abstraction. For example, the term $x^{*\multimap*}$, which is not in long form, can be applied to an argument of type $*$; in long form, this term becomes $\lambda y^*.x^{*\multimap*}y^*$, the possibility of applying it to a term of type $*$ is syntactically represented by the λ-abstraction. A term is in long normal form (lnf for short) if it is both in β-normal form and in long form. The set $\mathrm{lnf}_\Sigma^\alpha$ (*resp.* $\mathrm{clnf}_\Sigma^\alpha$) represents the set of terms of Λ_Σ^α in lnf (*resp.* the closed terms of Λ_Σ^α in lnf). In the sequel of the paper we only deal with terms in long form; thus each time we will write $\lambda\overrightarrow{x_S}.t$, $x\overrightarrow{t_S}$ or $a\overrightarrow{t_S}$, we will implicitly make the assumption that t, $x\overrightarrow{t_S}$ or $a\overrightarrow{t_S}$ has an atomic type.

We define homomorphisms between the higher-order signatures Σ_1 and Σ_2 to be pairs (f, g) such that f is a mapping from $\mathcal{T}_{\Sigma_1}$ to $\mathcal{T}_{\Sigma_2}$, and g is a mapping from Λ_{Σ_1} to Λ_{Σ_2}, and verifying:

1. if $\alpha \in \mathcal{A}_{\Sigma_1}$ then $f(\alpha) \in \mathcal{T}_{\Sigma_2}$, otherwise, $f(\alpha \multimap \beta) = f(\alpha) \multimap f(\beta)$
2. for all $a \in C_{\Sigma_1}$ such that $\tau_{\Sigma_1}(a) = \alpha$, $g(a) \in \mathrm{clnf}_{\Sigma_2}^{f(\alpha)}$
3. $g(x^\alpha) = x^{f(\alpha)}$

4. $g(t_1 t_2) = g(t_1)g(t_2)$
5. $g(\lambda x^{\alpha}.t) = \lambda x^{f(\alpha)}.g(t)$

One can easily check that whenever $t \in \Lambda^{\alpha}_{\Sigma_1}$, $g(t) \in \Lambda^{f(\alpha)}_{\Sigma_2}$. In general, given a homomorphism $\mathcal{L} = (f, g)$, we will write indistinctly $\mathcal{L}(\alpha)$ for $f(\alpha)$ and $\mathcal{L}(t)$ for $g(t)$. The *order* of $\mathcal{L}$ is $\max\{\mathrm{ord}(\mathcal{L}(\alpha)) | \alpha \in \mathcal{A}_{\Sigma_1}\}$.

An ACG (de Groote (2001)) is a 4-tuple $(\Sigma_1, \Sigma_2, \mathcal{L}, S)$ such that:

1. Σ_1 is a higher-order signature, *the abstract vocabulary*
2. Σ_2 is a higher-order signature, *the object vocabulary*
3. $\mathcal{L}$ is a homomorphism from Σ_1 to Σ_2, *the lexicon*
4. $S \in \mathcal{A}_{\Sigma_1}$

An *abstract constant* (*resp. object constant*) is an element of $\mathcal{C}_{\Sigma_1}$ (*resp.* $\mathcal{C}_{\Sigma_2}$), an *abstract type* (*resp. object type*) is an element of $\mathcal{T}_{\Sigma_1}$ (*resp.* $\mathcal{T}_{\Sigma_2}$). Given an abstract constant a, $\mathcal{L}(a)$ is called the *realization* of a.

An ACG $\mathcal{G} = (\Sigma_1, \Sigma_2, \mathcal{L}, S)$ defines two languages:

1. *the abstract language:* $\mathcal{A}(\mathcal{G}) = \mathrm{clnf}^{S}_{\Sigma_1}$
2. *the object language:* $\mathcal{O}(\mathcal{G}) = \{v \in \mathrm{clnf}_{\Sigma_2} | \exists t \in \mathcal{A}(\mathcal{G}).v =_{\beta\eta} \mathcal{L}(t)\}$

An ACG $\mathcal{G} = (\Sigma_1, \Sigma_2, \mathcal{L}, S)$ is said to be a *string ACG* if Σ_2 is a string signature and $\mathcal{L}(S) = *$. The *order of an ACG* is the order of its abstract signature.

9.3 Path in types, active substerms and active variables

We assume that we are given a signature Σ and that all the types and all the terms used in this section are built on that signature.

A linear λ-term $t \in \mathrm{lnf}^{\alpha}_{\Sigma}$ represents, via the Curry-Howard isomorphism, a cut-free proof of α in the *Intuitionnistic Implicative and Exponential Linear Logic*. This correspondence leads to a natural relation between subformulae of α and subterms of t. This section presents this relation which will play a central role in our encoding.

The subformulae of a type will be designated by means of paths. A path $\pi = i_1 \cdot i_2 \cdots i_{n-1} \cdot i_n$ is a possibly empty sequence of strictly positive integers; n is the length of π and when $n = 0$, π will be denoted by $\bullet$. Given a set of paths P, $i \cdot P$ denotes the set $\{i \cdot \pi | \pi \in P\}$. The set of paths in the type α, $\mathcal{P}_{\alpha}$ is defined as follows:

$$\mathcal{P}_{(\alpha_1,\ldots,\alpha_n)\multimap\alpha_0} = \{\bullet\} \cup \bigcup_{i=1}^{n} i \cdot \mathcal{P}_{\alpha_i} \text{ (recall that } \alpha_0 \text{ is atomic)}$$

The set $\mathcal{P}_{\alpha}$ is split within two parts: the positive paths, denoted by $\mathcal{P}^{+}_{\alpha}$ and the negative paths denoted by $\mathcal{P}^{-}_{\alpha}$. Positive (*resp.* negative) paths are the path of $\mathcal{P}_{\alpha}$ which have an even (*resp.* odd) length.

Given a path π, we define $p + \pi$ as: $p + \pi = \begin{cases} \bullet \text{ if } \pi = \bullet \\ (p + k) \cdot \pi' \text{ if } \pi = k \cdot \pi' \end{cases}$

Given $t \in \mathrm{lnf}^{\alpha}_{\Sigma}$, we define two particular subsets of $\mathcal{S}_t$, the set of *active subterms*, $\mathcal{AT}_t$, and the set of *active variables*, $\mathcal{AV}_t$. The sets $\mathcal{AT}_t$ and $\mathcal{AV}_t$ are defined as the smallest sets satisfying:

1. $([], t) \in \mathcal{AT}_t$
2. if $(C[], \lambda\overrightarrow{x_n}.t') \in \mathcal{AT}_t$ then for all $i \in [1, n]$,
$$(C[\lambda\overrightarrow{x_n}.C_{t',x_i}[]], x_i) \in \mathcal{AV}_t$$
3. if $(C[[]t_1 \ldots t_n], x) \in \mathcal{AV}_t$ then for all $i \in [1, n]$,
$$(C[xt_1 \ldots t_{i-1}[] \ldots t_n], t_i) \in \mathcal{AT}_t$$

If a term t can be applied to n arguments, then, given $t_1, \ldots, t_n$ terms in lnf, during the β-reduction of $tt_1 \ldots t_n$ the active variables of t will eventually substituted by a term during β-reduction and the residuals of the active subterms of t will eventually become the argument of a redex. On the other hand, the variables of t which are not active will never be substituted and the subterms of t which are not active will never be the argument of a redex.

We can now define two mutually recursive functions $\mathbf{AT}_t$ and $\mathbf{AV}_t$ respectively from $\mathcal{AT}_t$ onto $\mathcal{P}^+_\alpha$ and from $\mathcal{AV}_t$ onto $\mathcal{P}^-_\alpha$:

1. $\mathbf{AT}_t([], t) = \bullet$
2. if $\mathbf{AT}_t(C[], \lambda\overrightarrow{x_n}.t') = \pi$ then for all $i \in [1, n]$,
$$\mathbf{AV}_t(C[\lambda\overrightarrow{x_n}.C_{t',x_i}[]], x_i) = \pi \cdot i$$
3. if $\mathbf{AV}_t(C[[]t_1 \ldots t_n], x) = \pi$ then for all $i \in [1, n]$,
$$\mathbf{AT}_t(C[xt_1 \ldots t_{i-1}[] \ldots t_n], t_i) = \pi \cdot i$$

One can easily check that $\mathbf{AT}_t(C[], v) = \pi$ (*resp.* $\mathbf{AV}_t(C[], x) = \pi$) implies that the type of v (*resp.* x) is the type designated (in the obvious way) by π in α.

The functions $\mathbf{AT}_t$ and $\mathbf{AV}_t$ are bijections whose converse is $\mathbf{P}_t$:

1. $\mathbf{P}_t(\bullet) = ([], t)$
2. $\mathbf{P}_t(\pi \cdot i) = \begin{cases} (C[\lambda\overrightarrow{x_n}.C_{t',x_i}[], x_i) \text{ if } \mathbf{P}_t(\pi) = (C[], \lambda\overrightarrow{x_n}.t') \\ (C[xt_1 \ldots t_{i-1}[] \ldots t_n, t_i) \text{ if } \mathbf{P}_t(\pi) = (C[[]t_1 \ldots t_n], x) \end{cases}$

For all $(C[], t') \in \mathcal{AT}_t$ (*resp.* $(C[], x) \in \mathcal{AV}_t$) it is straightforward that $\mathbf{P}_t(\mathbf{AT}_t(C[], t')) = (C[], t')$ (*resp.* $\mathbf{P}_t(\mathbf{AV}_t(C[], x)) = (C[], x)$); and that for all $\pi \in \mathcal{P}^+_\alpha$ (*resp.* $\pi \in \mathcal{P}^-_\alpha$), $\mathbf{AT}_t(\mathbf{P}_t(\pi)) = \pi$ (*resp.* $\mathbf{AV}_t(\mathbf{P}_t(\pi)) = \pi$).

9.4 *h*-reduction

The DTWTs which encode second order string ACGs perform the normalization of the realization of abstract terms. They use a particular reduction strategy, *h-reduction*, which is related to *head linear reduction* (Danos and Regnier (2004)).

This reduction strategy is only defined for a particular class of λ-terms. Firstly, these λ-terms have to be built on a string signature Σ; secondly, they have a particular form. To describe this form, we need first define $\mathcal{N}_\Sigma^\alpha \subseteq \Lambda_\Sigma^\alpha$ ($\mathcal{N}_\Sigma = \bigcup_{\alpha\in\mathcal{T}_\Sigma} \mathcal{N}_\Sigma^\alpha$) as:

$$\mathcal{N}_\Sigma^\alpha ::= \mathrm{lnf}_\Sigma^\alpha \mid (\mathcal{N}_\Sigma^{\beta\multimap\alpha}\mathcal{N}_\Sigma^\beta)$$

Then, the set of terms we are interested in are the HT-terms defined by the following grammar:

$$\mathcal{HT} ::= \mathcal{N}_\Sigma^* \mid c\mathcal{HT} \mid (\lambda x_1^{\alpha_1}\dots x_n^{\alpha_n}.\mathcal{HT})\mathcal{N}_\Sigma^{\alpha_1}\dots\mathcal{N}_\Sigma^{\alpha_n}$$

where $c \in \mathcal{C}_\Sigma$. Every HT-term is in Λ_Σ^* and is of the form:

$$(\lambda\overrightarrow{x_{S_1}}.\overrightarrow{c_{T_1}}(\dots(\lambda\overrightarrow{x_{S_n}}.\overrightarrow{c_{T_n}}(x_j\overrightarrow{t_Q}))\overrightarrow{v_{S_n}}\dots))\overrightarrow{v_{S_1}}$$

so that $S_i \cap S_j \neq \emptyset$ implies that $i = j$, v_k (with $k \in \bigcup_{i\in[1,n]} S_i$) and t_q (with $q \in Q$) are elements of $\mathcal{N}_\Sigma$.

Given a HT-term,

$$t = (\lambda\overrightarrow{x_{S_1}}.\overrightarrow{c_{T_1}}(\dots(\lambda\overrightarrow{x_{S_n}}.\overrightarrow{c_{T_n}}(x_j\overrightarrow{t_Q}))\overrightarrow{v_{S_n}}\dots))\overrightarrow{v_{S_1}}$$

we say that t h-contracts to t' (noted $t \rightarrow_h t'$) if

$$t' = (\lambda\overrightarrow{x_{S'_1}}.\overrightarrow{c_{T_1}}(\dots(\lambda\overrightarrow{x_{S'_n}}.\overrightarrow{c_{T_n}}(v_j\overrightarrow{t_Q}))\overrightarrow{v_{S'_n}}\dots))\overrightarrow{v_{S'_1}}$$

where $S'_k = S_k\backslash\{j\}$. It is a routine to check that $t =_\beta t'$, that t' is also a HT-term and that the normal form of t can be obtained in a finite number of h-contractions. The reflexive and transitive closure of $\rightarrow_h$, *h-reduction*, will be written $\stackrel{*}{\rightarrow}_h$.

Given $\mathcal{G} = (\Sigma_1, \Sigma_2, S, \mathcal{L})$ a second order string ACG, and $u \in \mathrm{clnf}_\Sigma^S$, we are going to see how h-contraction normalizes $\mathcal{L}(u)$. The determinism of $\rightarrow_h$ allows one to predict statically (*i.e.* without performing the reduction) which subterm of $\mathcal{L}(u)$ will be substituted to a given bound variable in $\mathcal{L}(u)$ during h-reduction. This prediction is based on the notions of *replaceable variables* and *unsafe terms* introduced by Böhm and Dezani-Ciancaglini (1975). Replaceable variables and unsafe terms of u belong to $\mathcal{S}_{\mathcal{L}(u)}$ and will be respectively denoted by $\mathcal{RV}_u$ and $\mathcal{UT}_u$.

If $(C[], a) \in \mathcal{S}_u$ and $(C'[], x) \in \mathcal{AV}_{\mathcal{L}(a)}$, then $(\mathcal{L}(C)[C'[]], x) \in \mathcal{RV}_u$; $\mathcal{UT}_u$ is the smallest set verifying:

1. if $(C[], a\overrightarrow{v_{\rho_a}}) \in \mathcal{S}_u$ and $C[] \neq []$ then $(\mathcal{L}(C)[], \mathcal{L}(a\overrightarrow{v_{\rho_a}})) \in \mathcal{UT}_u$
2. if $(C[], a) \in \mathcal{S}_u$ and $(C'[], v) \in \mathcal{AT}_{\mathcal{L}(a)}$ then $(\mathcal{L}(C)[C'[]], v) \in \mathcal{UT}_u$

The prediction will be given by ϕ_u, a bijection between $\mathcal{RV}_u$ and $\mathcal{UT}_u$. The definition of ϕ_u relies on few more technical definitions.

Given $(C_a[], a) \in \mathcal{S}_u$ such that $C_a[] = C[[]v_1\dots v_{\rho_a}]$, then

$$(C[av_1\dots v_{i-1}[]\dots v_{\rho_a}], v_i)$$

is the i^{th} *argument* of $(C_a[], a)$. Given $(C_a[], a), (C_b[], b) \in \mathcal{S}_u$, we say that $(C_a[], a)$ is the *head of the i^{th} argument* of $(C_b[], b)$ if

$$C_b[] = C[[]v_1 \dots v_{i-1}(a\overrightarrow{w_{\rho_a}}) \dots v_{\rho_b}] \text{ and } C_a[] = C[bv_1 \dots v_{i-1}([]\overrightarrow{w_{\rho_a}}) \dots v_{\rho_b}]$$

Given $(C[], x) \in \mathcal{RV}_u$, we now define $\phi_u(C[], x)$. As $(C[], x) \in \mathcal{RV}_u$, we have $(C_a[], a) \in \mathcal{S}_u$ and $C_x[]$ such that $(C_x[], x) \in \mathcal{AV}_{\mathcal{L}(a)}$ and $C[] = \mathcal{L}(C_a)[C_x[]]$. Let $\pi = \mathbf{AV}_{\mathcal{L}(a)}(C_x[], x)$, since $\pi \in \mathcal{P}^{-}_{\mathcal{L}(\tau_{\Sigma_1}(a))}$, π is of odd length, and $\pi = i.\pi'$. Then we have three cases:

1. if $i \leq \rho_a$ and $\pi' = \bullet$, then $\phi_u(C[], x) = (\mathcal{L}(C')[], \mathcal{L}(t))$ where $(C'[], t)$ is the i^{th} argument of $(C_a[], a)$
2. if $i \leq \rho_a$ and $\pi' \neq \bullet$, then $\phi_u(C[], x) = (\mathcal{L}(C_b)[C'[]], t)$ where $(C_b[], b)$ is the head of the i^{th} argument of $(C_a[], a)$ and $(C'[], t) = \mathbf{P}_{\mathcal{L}(b)}(\rho_b + \pi')$
3. if $i > \rho_a$ then $\phi_u(C[], x) = (\mathcal{L}(C_b)[C'[]], t)$ where $(C_a[], a)$ is the head of the k^{th} argument of $(C_b[], b)$ and $(C'[], t) = \mathbf{P}_{\mathcal{L}(b)}(k \cdot (i - \rho_a) \cdot \pi')$.

Computing $\phi_u(C[], x)$ only requires to know about the immediate surrounding of a. This is the reason why the normalization of $\mathcal{L}(u)$ can be performed by a DTWT. To prove the correctness of the prediction of ϕ_u we need the notion of *strict residual*: given t and t' such that $t \overset{*}{\rightarrow}_h t'$, $(C[], v) \in \mathcal{S}_t$ and $(C'[], v) \in \mathcal{S}_{t'}$, we say that $(C'[], v)$ is *the strict residual* of $(C[], v)$ whenever $C[xy_1 \dots y_n] \overset{*}{\rightarrow}_h C'[xy_1 \dots y_n]$ with $FV(v) = \{y_1, \dots, y_n\}$ and x is a fresh variable.

Given t such that $\mathcal{L}(u) \overset{*}{\rightarrow}_h t$, we say that t is *predicted by* ϕ_u if the two following properties hold:

1. for all $(C[], (\lambda\overrightarrow{x_n}\lambda\overrightarrow{y_q}.v)\overrightarrow{v_n}) \in \mathcal{S}_t$ and $i \in [1, n]$, the fact that
$$(C[(\lambda\overrightarrow{x_n}\lambda\overrightarrow{y_q}.C_{v,x_i}[])\overrightarrow{v_n}], x_i)$$
is the strict residual of $(C_{x_i}[], x_i) \in \mathcal{RV}_{\mathcal{L}(u)}$ implies that
$$(C[(\lambda\overrightarrow{x_n}\lambda\overrightarrow{y_q}.v)v_1 \dots v_{i-1}[] \dots v_n], v_i)$$
is the strict residual of $\phi_u(C_{x_i}[], x_i)$.
2. for all $(C[[]\overrightarrow{v_q}], x) \in \mathcal{S}_t$, $(C[[]\overrightarrow{v_q}], x)$ is the strict residual of some $(C'[[]\overrightarrow{v_q}], x) \in \mathcal{RV}_u$.

We are now going to show that h-reduction preserves the predictions of ϕ_u. This will be achieved by using the following technical lemma:

Lemma 22 *Given $(C[[]\overrightarrow{v_q}], x) \in \mathcal{RV}_u$ if we have $\phi_u(C[[]\overrightarrow{v_q}], x) = (C'[], t')$ then $t' = (\lambda\overrightarrow{x_p}\overrightarrow{y_q}.w)\overrightarrow{w_p}$ and we have*

$$\phi_u(C'[(\lambda\overrightarrow{x_p}\overrightarrow{y_q}.C_{w,y_k}[])\overrightarrow{w_p}], y_k) = (C[xv_1 \dots v_{k-1}[] \dots v_q], v_k)$$

Proof. This proof only consists in unfolding the definitions. Since $(C[[]\overrightarrow{v_q}], x) \in \mathcal{RV}_u$, we must have $(C_a[], a) \in \mathcal{S}_u$ and $C_x[]$ such that:

1. $C[[]\overrightarrow{v_q}] = \mathcal{L}(C_a)[C_x[[]\overrightarrow{v_q}]]$
2. $(C_x[[]\overrightarrow{v_q}], x) \in \mathcal{AV}_{\mathcal{L}(a)}$
3. $\mathbf{AV}_{\mathcal{L}(a)}(C_x[[]\overrightarrow{v_q}], x) = i \cdot \pi$ for some i and π

There are three different cases depending on i and π.

Case 1: $i \leq \rho_a$ and $\pi = \bullet$: this case is very similar to the following one and is thus left to the reader. It is the only case where p may be different from 0.

Case 2: $i \leq \rho_a$ and $\pi \neq \bullet$: by definition if $(C_b[], b)$ is the head of the i^{th} argument of $(C_a[], a)$, and if $\mathbf{P}_{\mathcal{L}(b)}(\rho_b + \pi) = (C''[], \lambda\overrightarrow{y_q}.w)$ then $C'[] = \mathcal{L}(C_b)[C''[]]$ and $t' = \lambda\overrightarrow{y_q}.w$. Let's now suppose that $\pi = m \cdot \pi'$, then we have that $\mathbf{AV}_{\mathcal{L}(b)}(\lambda\overrightarrow{y_q}.C_{w,y_k}[], y_k) = (\rho_b + \pi) \cdot k = (\rho_b + m) \cdot \pi' \cdot k$. Therefore, as $\rho_b + m > \rho_b$ and as $(C_b[], b)$ is the head of the i^{th} argument of $(C_a[], a)$, we have that $\phi_u((\lambda\overrightarrow{y_q}.C_{w,y_k}[]), y_k) = (\mathcal{L}(C_a)[C_k[]], u_k)$ where

$$(C_k[], u_k) = \mathbf{P}_{\mathcal{L}(a)}(i \cdot (\rho_b + m - \rho_b) \cdot \pi' \cdot k) = \mathbf{P}_{\mathcal{L}(a)}(i \cdot \pi \cdot k)$$

But we have that $\mathbf{AV}_{\mathcal{L}(a)}(C_x[[]\overrightarrow{v_q}], x) = i \cdot \pi$ which implies that

$$(C_k[], u_k) = \mathbf{P}_{\mathcal{L}(a)}(i \cdot \pi \cdot k) = (C_x[xv_1 \dots v_{k-1}[] \dots v_r], v_k).$$

Finally as $C[] = \mathcal{L}(C_a)[C_x[]]$ we get the result.

Case 3: $i > \rho_a$: this case is similar to the previous one. ☐

Proposition 23 *If $\mathcal{L}(u) \xrightarrow{*}_h t$, then t is predicted by ϕ_u.*

Proof. This proof is done by induction on the number of h-contraction steps of the reduction. The case where this is zero is a simple application of the definitions. Now let's suppose that $\mathcal{L}(u) \xrightarrow{*}_h t \rightarrow_h t'$, then, by induction hypothesis, t is predicted by ϕ_u; furthermore, t is a HT-term, thus

$$t = (\lambda\overrightarrow{x_{S_1}}.\overrightarrow{c_{T_1}}(\dots(\lambda\overrightarrow{x_{S_n}}.\overrightarrow{c}_{T_n}(x_j\overrightarrow{t_Q}))\overrightarrow{v_{S_n}}\dots))\overrightarrow{v_{S_1}}$$

and

$$t' = (\lambda\overrightarrow{x_{S'_1}}.\overrightarrow{c_{T_1}}(\dots(\lambda\overrightarrow{x_{S'_n}}.\overrightarrow{c}_{T_n}(v_j\overrightarrow{t_Q}))\overrightarrow{v_{S'_n}}\dots))\overrightarrow{v_{S'_1}}$$

with $S'_i = S_i \backslash \{j\}$.

Within the two conditions required to obtain that t' is predicted by ϕ_u, only the first one requires more than a straightforward application of the induction hypothesis. There is actually only one subterm of t' which is problematic: $v_j\overrightarrow{t_Q}$. From the induction hypothesis we know that the subterm corresponding to x_j in t is the strict residual of $(C[[]\overrightarrow{t_Q}], x_j) \in \mathcal{RV}_u$ and that the subterm corresponding to v_j in t is the strict residual of $\phi_u(C[[]\overrightarrow{t_Q}], x_j)$. Finally the previous lemma allows us to conclude that $v_j\overrightarrow{t_Q}$ fullfills the first condition. ☐

9.5 Encoding second order string ACGs with DTWT

We are now going to show how to encode second order string ACGs into DTWT. We do not follow the standard definition of DTWT as given in Aho and Ullman (1971). Indeed, instead of walking on the parse trees of a context free grammar, the transducers we use walk on linear λ-terms built on a second order signature. But, as these sets of λ-terms are isomorphic to regular sets of trees, the string languages outputed by our transducers are the same as those of usual DTWT. By abuse, we call our transducers DTWT.

A DTWT is defined as a 6-tuple

$$\mathbf{A} = (\Sigma, D, Q, T, \delta, \mathbf{q}_0, \mathbf{q}_f)$$

where Σ is a second order signature; $D \in \mathcal{A}_\Sigma$; Q is a finite set of states; T is a finite set of terminals; δ , the transition function, is a partial function from $C_\Sigma \times (Q \setminus \{\mathbf{q}_f\})$ to $(\{up; stay\} \cup (down \times \mathbb{N}^+)) \times Q \times T^*$ where $\mathbb{N}^+$ denotes the set of strictly positive natural numbers and T^* denotes the monoid freely generated by T; $\mathbf{q}_0 \in Q$ is the initial state; and $\mathbf{q}_f \in Q$ is the final state. A *configuration* of $\mathbf{A}$ is given by $(C[], a, \mathbf{q}, s)$ where $C[a] \in \text{clnf}_\Sigma^D$, $a \in C_\Sigma$, $\mathbf{q} \in Q$ and $s \in T^*$; *initial configurations* are of the form $([]\overrightarrow{v_{\rho_a}}, a, \mathbf{q}_0, \epsilon)$ (ϵ being the empty string) where $a\overrightarrow{v_{\rho_a}} \in \text{clnf}_\Sigma^D$. The automaton $\mathbf{A}$ defines a move relation, $\vdash_\mathbf{A}$ ($\vdash_\mathbf{A}^*$ is the reflexive transitive closure of $\vdash_\mathbf{A}$), between configurations: $(C[], a, \mathbf{q}, s) \vdash_\mathbf{A} (C'[], b, \mathbf{q}', sw)$ if $\delta(a, \mathbf{q}) = (\mathbf{q}', m, w)$ and one of the following holds:

1. $m = up$ and $(C[], a)$ is the head of one of the arguments of $(C'[], b)$
2. $m = stay$ and $(C'[], b) = (C[], a)$
3. $m = (down, i)$ and $(C'[], b)$ is the head of the i^{th} argument of $(C[], a)$

Given $a\overrightarrow{v_{\rho_a}} \in \text{clnf}_\Sigma^D$, $a\overrightarrow{v_{\rho_a}}$ generates s with $\mathbf{A}$ if

$$([]\overrightarrow{v_{\rho_a}}, a, \mathbf{q}_0, \epsilon) \vdash_\mathbf{A}^* (C[], b, \mathbf{q}_f, s).$$

The language of $\mathbf{A}$, $\mathbb{L}_\mathbf{A}$, is $\{s | \exists v \in \text{clnf}_\Sigma^D . v \text{ generates } s\}$.

Given a second order string ACG $\mathcal{G} = (\Sigma_1, \Sigma_2, \mathcal{L}, S)$ we are going to build an automaton $\mathbf{A}_\mathcal{G} = (\Sigma, D, Q, T, \delta, \mathbf{q}_0, \mathbf{q}_f)$ such that $\mathcal{O}(\mathcal{G}) = \{/w/ | w \in \mathbb{L}_{\mathbf{A}_\mathcal{G}}\}$. Let $k_\mathcal{G} = \max\{\rho_a | a \in C_{\Sigma_1}\}$, we then define Σ as:

1. $\mathcal{A}_\Sigma = \mathcal{A}_{\Sigma_1} \times [1, k_\mathcal{G}]$
2. $C_\Sigma = C_{\Sigma_1} \times [1, k_\mathcal{G}]$
3. if $\tau_{\Sigma_1}(a) = (\alpha_1, \ldots, \alpha_n) \multimap \alpha$ then

$$\tau_\Sigma((a, k)) = ((\alpha_1, 1), \ldots, (\alpha_n, n)) \multimap (\alpha, k).$$

Remark that if $v \in \text{clnf}_\Sigma^{(\alpha,k)}$, then for all $(C[], (a, j)) \in \mathcal{S}_v$, $C[] \neq []\overrightarrow{v_{\rho_a}}$ implies that $(C[], (a, j))$ is the head of the j^{th} argument of $(C'[], (b, l)) \in \mathcal{S}_v$. Furthermore, given $v = (a, k)\overrightarrow{v_{\rho_a}} \in \text{clnf}_\Sigma^{(\alpha,k)}$ we note $\widetilde{v}$ the term of $\text{clnf}_{\Sigma_1}^\alpha$ such that $\widetilde{v} = a\overrightarrow{\widetilde{v_{\rho_a}}}$.

Then $D = (S, 1)$, $Q = ([0, k_{\mathcal{G}}] \times P) \cup \{\mathbf{q}_f\}$ where $P = \bigcup_{\alpha \in C_{\Sigma_1}} \mathcal{P}_{\mathcal{L}(\alpha)}$, $\mathbf{q}_0 = (0, \bullet)$; building δ requires some more definitions.

Given (a, k) and (i, π), the *selection path* of (a, k) and (i, π) is:

$$\pi' = \begin{cases} i \cdot \pi \text{ if } i > 0 \\ \rho_a + \pi \text{ if } i = 0 \end{cases}$$

If the selection path of (a, k) and (i, π) is in $\mathcal{P}^+_{\mathcal{L}(\tau_{\Sigma_1}(a))}$ then we say that (a, k) and (i, π) are *coherent*; δ will be only defined on coherent pairs of (a, k) and (i, π). A configuration $K = (C[], (a, k), (i, \pi), w)$ is said to be *coherent* if (a, k) and (i, π) are coherent.

If (a, k) and (i, π) are coherent and if π' is their selection path, then we define the *focused term* of (a, k) and (i, π) as $\mathbf{P}_{\mathcal{L}(a)}(\pi')$. Furthermore, if $(C[], t)$ is the focused term of (a, k) and (i, π) and if $t = \lambda\overrightarrow{x_p}.\overrightarrow{c_n}(x\overrightarrow{v_q})$, then $(C[\lambda\overrightarrow{x_p}.\overrightarrow{c_n}([]\overrightarrow{v_q})], x))$ is called the *focused variable* of (a, k) and (i, π).

If (a, k) and (i, π) are coherent then $\delta((a, k), (i, \pi)) = (\mathbf{q}, move, w)$ depends on the focused term of (a, k) and (i, π), (noted $(C[], t)$):

1. if $t = \overrightarrow{c_n}\#$ then $\mathbf{q} = \mathbf{q}_f$, $move = stay$ and $w = c_1 \dots c_n$
2. if $t = \lambda\overrightarrow{x_p}.\overrightarrow{c_n}(x\overrightarrow{v_q})$, $\mathbf{AV}_{\mathcal{L}(a)}(C[\lambda\overrightarrow{x_p}.\overrightarrow{c_n}([]\overrightarrow{v_q})], x) = l \cdot \pi''$ and $l > \rho_a$ then $\mathbf{q} = (k, (l - \rho_a) \cdot \pi'')$, $move = up$ and $w = c_1 \dots c_n$
3. if $t = \lambda\overrightarrow{x_p}.\overrightarrow{c_n}(x\overrightarrow{v_q})$, $\mathbf{AV}_{\mathcal{L}(a)}(C[\lambda\overrightarrow{x_p}.\overrightarrow{c_n}([]\overrightarrow{v_q})], x) = l \cdot \pi''$ and $l \leq \rho_a$ then $\mathbf{q} = (0, \pi'')$, $move = (down, l)$ and $w = c_1 \dots c_n$

We now relate the walk of $\mathbf{A}_{\mathcal{G}}$ on $v \in \mathrm{clnf}^{(S,1)}_{\Sigma}$ with the h-reduction of $\mathcal{L}(\widetilde{v})$. To establish this relation we need to show that the transducer computes $\phi_{\widetilde{v}}$. Given a coherent configuration $K = (C[], (a, k), (i, \pi), w)$, the *activated term* of K is $(\mathcal{L}(C')[], \mathcal{L}(a\overrightarrow{\widetilde{v_{\rho_a}}}))$ if $(i, \pi) = (0, \bullet)$ and $\widetilde{C}[] = C'[[]\overrightarrow{\widetilde{v_{\rho_a}}}]$, otherwise it is $(\mathcal{L}(\widetilde{C})[C'[]], t)$ if $(C'[], t)$ is the focused term of (a, k) and (i, π); the *activated variable* of K is $(\mathcal{L}(\widetilde{C})[C'[]], x)$ if the focused variable of (a, k) and (i, π) is $(C'[], x)$. We will show that given K_1 and K_2 such that $K_1 \vdash_{\mathbf{A}_{\mathcal{G}}} K_2$, if $(C[], x)$ is the activated variable of K_1 then $\phi_{\widetilde{v}}(C[], x)$ is the activated term of K_2. This property shows that $\mathbf{A}_{\mathcal{G}}$ performs the h-reduction of $\mathcal{L}(\widetilde{v})$ and that if $\mathcal{L}(\widetilde{v})$ normalizes to $/w/$ then, walking on v, $\mathbf{A}_{\mathcal{G}}$ ends in the final state and outputs w.

Lemma 24 *Given $v = (a, 1)\overrightarrow{v_{\rho_a}} \in \mathrm{clnf}^{(S,1)}_{\Sigma}$ and two coherent configurations K_1 and K_2 such that $([]\overrightarrow{v_{\rho_a}}, (a, 1), (0, \bullet), \epsilon) \vdash^*_{\mathbf{A}_{\mathcal{G}}} K_1 \vdash_{\mathbf{A}_{\mathcal{G}}} K_2$, if $(C[], x)$ is the activated variable of K_1 then $\phi_{\widetilde{v}}(C[], x)$ is the activated term of K_2.*

Proof. As for the proof of lemma 22, this proof is mainly based on the unfolding of the definitions. We simply compute $\phi_{\widetilde{v}}(C[], x)$ and the activated term of K_2 and then show that they are the same.

We assume that $K_r = (C_r[], (a_r, k_r), (i_r, \pi_r), w_r)$ with $r \in [1, 2]$, that π'_r is the selection path of K_r. If $\mathbf{P}_{\mathcal{L}(a_1)}(\pi'_1) = (C'_1[], \lambda\overrightarrow{x_p}.\overrightarrow{c_n}(x\overrightarrow{v_q}))$, then let $\pi''_1 =$

$\mathbf{AV}_{\mathcal{L}(a_1)}(C'_1[\lambda\overrightarrow{x_p}.\overrightarrow{c_n}([]\overrightarrow{v_q}), x)$; as $\pi''_1 \in \mathcal{P}^-_{\mathcal{L}(\tau_{\Sigma_1}(a_1))}$, we know that $\pi''_1 = i \cdot \pi''$. We then have three cases:

Case 1: if $i \leq \rho_{a_1}$ and $\pi'' = \bullet$, then $\phi_{\widetilde{v}}(C[], x) = (\mathcal{L}(C')[], \mathcal{L}(t))$ if $(C'[], t)$ is the i^{th} argument of $(C_1[], a_1)$. But in that case, we have that $\delta((a_1, k_1), (i_1, \pi_1)) = ((0, \bullet), (down, i), c_1 \ldots c_n)$; thus (a_2, k_2) is the head of the i^{th} argument of (a_1, k_1) and as $(i_2, \pi_2) = (0, \bullet)$, we obtain, by definition, that the activated term of K_2 is indeed $(\mathcal{L}(C')[], \mathcal{L}(t))$.

Case 2: if $i \leq \rho_{a_1}$ and $\pi'' \neq \bullet$, then $\phi_{\widetilde{v}}(C[], x) = (\mathcal{L}(C_b)[C'[]], t)$ if $(C_b[], b)$ is the i^{th} argument of $(C_1[], a_1)$ and if $(C'[], t) = \mathbf{P}_{\mathcal{L}(b)}(\rho_b + \pi'')$. In that case, we have $\delta((a_1, k_1), (i_1, \pi_1)) = ((0, \pi''), (down, i), c_1 \ldots c_n)$; therefore, (a_2, k_2) is the head of i^{th} argument of (a_1, k_1) which implies that $(C_2[], a_2) = (C_b[], b)$; finally by definition we have that the activated term of K_2 is $(\mathcal{L}(C_b)[C'[]], t) = \phi_{\widetilde{v}}(C[], x)$.

Case 3: if $i > \rho_{a_1}$ then $\phi_{\widetilde{v}}(C[], x) = (\mathcal{L}(C_b)[C'[]], t)$ if $(C_{a_1}[], a_1)$ is the head of the ${k_1}^{\text{th}}$ argument of $(C_b[], b)$ and $(C'[], t) = \mathbf{P}_{\mathcal{L}(b)}(k_1 \cdot (i - \rho_{a_1}) \cdot \pi'')$. In that case, we have $\delta((a_1, k_1), (i, \pi)) = ((k_1, (i - \rho_{a_1}) \cdot \pi''), up, c_1 \ldots c_n)$, and the definition leads to the fact that the activated term of K_2 is $(\mathcal{L}(C_b)[C'[]], t) = \phi_{\widetilde{v}}(C[], x)$.
$\square$

Proposition 25 *Given* $u \in \mathrm{clnf}^S_{\Sigma_1}$, *there is a unique* $v = (a, 1)\overrightarrow{v_{\rho_a}} \in \mathrm{clnf}^{(S,1)}_{\Sigma}$ *such that* $\widetilde{v} = u$, *and* $([]\overrightarrow{v_{\rho_a}}, (a, 1), (0, \bullet), \epsilon) \vdash^*_{\mathbf{A}_{\mathcal{G}}} (C[], b, q_f, w)$ *iff* $\mathcal{L}(u) =_{\beta\eta} /w/$.

Proof. The existence and the uniqueness of v are obvious from the definition of Σ. To prove the proposition it suffices to study the walk of $\mathbf{A}_{\mathcal{G}}$ on v and the h-reduction of $\mathcal{L}(u)$ in parallel: assume that $K_1 = ([]\overrightarrow{v_{\rho_a}}, (a, 1), (0, \bullet), \epsilon)$, $t_1 = \mathcal{L}(u)$, $K_1 \vdash^k_{\mathbf{A}_{\mathcal{G}}} K_k$ and $t_1 \xrightarrow{k}_h t_k$ (where $\vdash^k_{\mathbf{A}_{\mathcal{G}}}$ corresponds to k steps of $\mathbf{A}_{\mathcal{G}}$ and $\xrightarrow{k}_h$ to k steps of h-reduction). The use of the previous lemma and an induction on k prove that t_k is of the form

$$t_k = (\lambda\overrightarrow{x_{S_1}}.\overrightarrow{c_{T_1}}(\ldots(\lambda\overrightarrow{x_{S_k}}.\overrightarrow{c_{T_k}}(x_j\overrightarrow{t_Q}))\overrightarrow{v_{S_k}}\ldots))\overrightarrow{v_{S_1}}$$

if and only if $K_k = (C_k[], (a^k, l_k), (i_k, \pi_k), w_k)$ so that $w_k = \overrightarrow{c_{T_1}} \ldots \overrightarrow{c_{T_{k-1}}}$, if $(C'_k[], \lambda\overrightarrow{x_{S_k}}.\overrightarrow{c_{T_k}}(x_j\overrightarrow{t_Q})) \in \mathcal{S}_{t_k}$ (with the obvious $C'_k[]$) is the strict residual of $(C''_k[], \lambda\overrightarrow{x_{S_k}}.\overrightarrow{c_{T_k}}(x_j\overrightarrow{t_Q})) \in \mathcal{S}_{t_1}$ then $(C''_k[], \lambda\overrightarrow{x_{S_k}}.\overrightarrow{c_{T_k}}(x_j\overrightarrow{t_Q}))$ is the activated term of K_k and $(C''_k[\lambda\overrightarrow{x_{S_k}}.\overrightarrow{c_{T_k}}([]\overrightarrow{t_Q})], x_j)$ is the activated variable of K_k. This allows us to conclude that the walk ends in the configuration $(C[], b, q_f, w)$ iff $\mathcal{L}(u) =_{\beta\eta}$ $/w/$. $\square$

This finally shows that $\mathcal{O}(\mathcal{G})$ is indeed equal to $\{/w/ | w \in \mathbb{L}_{\mathbf{A}_{\mathcal{G}}}\}$.

9.6 Conclusions and future work

In this paper, we have proved that the languages defined by second order string ACGs were the same as the output languages of DTWT. From the results of Weir (1992) and de Groote and Pogodalla (2004), we obtain as a corolarry that the languages defined by second order string ACGs are exactly the languages defined by LCFRS. Furthermore as, according to de Groote and Pogodalla (2004), LCFRS can be encoded by second order string ACGs with a fourth order lexicons, we obtain that every second order string ACG can be encoded by another one whose lexicon has at most fourth order.

In our next work, we would like to exhibit a direct translation of a second order string ACG into another one with a fourth order lexicon. This would help understanding how relevant the order of the lexicon is. We conjecture that using lexicons of order greater than four may lead to more compact grammars. The problem is to know how compact those grammars can be and if the compaction is important whether it can be used do design large grammars for natural languages.

As the tools we used are general, we think it is possible to prove that any second order ACG can be represented as a second order ACG whose lexicon is at most fourth order. Indeed, the notion of paths and the relations they establish with active subterms and active variables do not depend on the problem. The only definition which is dependant of the fact we deal with strings is the definition of h-reduction. We nevertheless think that, provided we define a generalized notion of DTWT which would output linear λ-terms instead of strings, we can show that second order ACGs can be encoded with these generalized DTWTs. It would remain to encode those DTWTs with second order ACGs with a fourth order lexicon to generalize our result. But this last part does not seem too difficult.

The first part seems also feasible since it should be possible to generalize h-reduction. Indeed, instead of having a unique variable on which we could make the substitution, the fact that the constants in the term introduce some branching may lead to have several such variables. This would correspond on the generalized DTWTs to the fact that when it would output a branching constant the transducer should duplicate its head in order to have one head to generate each argument of that constant.

Finally this work may lead to the definition of an abstract machine for second order ACGs. Such a machine would be valuable to study the problem of parsing second order ACGs and give insights on the strategies that can be implemented for those grammars. Furthermore, as such a machine would have a language made of linear λ-terms, it would be a first step towards the definition of an abstract machine whose language is a set of λ-terms. In Montague style semantics, the problem of generation mainly consists in parsing languages of λ-terms. We would then obtain a valuable tool to study the problem of generation in that setting.

References

Aho, A. V. and J. D. Ullman. 1971. Translations on a context free grammar. *Information and Control* 19(5):439–475.

Barendregt, Henk P. 1984. *The Lambda Calculus: Its Syntax and Semantics*, vol. 103. Studies in Logic and the Foundations of Mathematics, North-Holland Amsterdam. revised edition.

Böhm, Corrado and Mariangiola Dezani-Ciancaglini. 1975. Lambda-terms as total or partial functions on normal forms. In C. Böhm, ed., *Lambda-Calculus and Computer Science Theory*, vol. 37 of *Lecture Notes in Computer Science*, pages 96–121. Springer. ISBN 3-540-07416-3.

Danos, Vincent and Laurent Regnier. 2004. How abstract machines implement head linear reduction. Preprint of the Institut de Mathématiques de Luminy.

de Groote, Philippe. 2001. Towards abstract categorial grammars. In A. for Computational Linguistic, ed., *Proceedings 39th Annual Meeting and 10th Conference of the European Chapter*, pages 148–155. Morgan Kaufmann Publishers.

de Groote, Philippe and Sylvain Pogodalla. 2004. On the expressive power of abstract categorial grammars: Representing context-free formalisms. *Journal of Logic, Language and Information* 13(4):421–438.

Girard, Jean-Yves. 1987. Linear logic. *Theoretical Computer Science* 50:1–102.

Huet, Gérard. 1997. The zipper. *Journal of Functional Programming* 7(5):549–554.

Montague, Richard. 1974. *Formal Philosophy: Selected Papers of Richard Montague*. Yale University Press, New Haven, CT.

Salvati, Sylvain. 2005. *Problèmes de filtrage et problèmes d'analyse pour les grammaires catégorielles abstraites*. Ph.D. thesis, Institut National Polytechnique de Lorraine.

Weir, David Jeremy. 1988. *Characterizing mildly context-sensitive grammar formalisms*. Ph.D. thesis, University of Pennsylvania, Philadephia, PA. Supervisor-Aravind K. Joshi.

Weir, David J. 1992. Linear context-free rewriting systems and deterministic tree-walking transducers. In *ACL*, pages 136–143.

Yoshinaka, Ryo and Makoto Kanazawa. 2005. The complexity and generative capacity of lexicalized abstract categorial grammars. In *LACL*, pages 330–346.

10

Sidewards without copying

Edward P. Stabler

A traditional movement step relates a single source position to a single c-commanding target position, and never moves an argument to another argument position. But head movement involves non-c-command relations, and control relates two argument positions that are not always in a c-command relation. Special mechanisms could be invoked for these things, but a different strategy slightly generalizes movement and enforces certain fundamental symmetries observed by all movements to block overgeneration. This paper defines a class of ‘sideward movement grammars’ (SMMGs) with such symmetries, with example applications to adjunct control and head movement. These grammars allow copying, but the question of whether to copy is completely independent of the question of whether to allow sideward movement. Furthermore, since these grammars distinguish complement attachments from others, a simple CED-like constraint can block extractions from specifiers and adjuncts except in the exceptional circumstance of adjunct control. SMMG definable languages are all PMCFG definable, and hence are efficiently recognizable.

10.1 Introduction

One of the most basic properties of human language is its simple, recursive, layered character in which similar structure is iterated, sometimes with special variations at the top, matrix level and at the deepest levels:

Does Alice know that	Bob thinks that	Carol says	you like her?
3	2	1	0

FG-2006.
Edited by Paola Monachesi, Gerald Penn,
Giorgio Satta and Shuly Wintner.

Certain kinds of recursive symmetry in languages allow the 'pumping lemmas' which have been valuable diagnostics of the availability of certain kinds of grammars. A regular grammar for a language is only possible when the language has a simple symmetry of this kind; context free grammars have a weaker requirement, and so on through the hierarchy of multiple context free languages (Seki et al., 1991), etc.

Many descriptions of human languages involve rearranging constituents. In grammars with movements, how is the structure of each 'layer' affected? This fundamental question is a topic of active study. In early transformational grammars, a set of base structures is generated and then transformed into surface structures, as in the following example (with *e* and *t* unpronounced):

[I [know [*e* [I [*e* [saw [who]]]]]]] ⟶ [I [know [who [I [*t* [saw [*t*]]]]]].

The sequences of positions related by movement in these accounts are not random. Among other things, landing sites of movement do not disrupt layer structure too much ('structure preservation', 'shape conservation'), and when an element moves through several clauses, it never moves from a high position in a lower clause to a lower position in a higher clause (cf. the 'ban on improper movement' 'chain uniformity', 'level embedding'). So in effect, the hierarchy of each layer of phrase structure is respected in sequences of movements too, another reflection of the basic invariants mentioned at the outset.

Some recent grammars compose generation and transformation steps,[1] so transformations are, in effect, executed as soon as requisite structure is built, reducing the need for revising completed structure:

1. [saw]+[who] $\xrightarrow{\text{merge}}$ [saw [who]]
2. [saw [who]]+[I] $\xrightarrow{\text{merge}}$ [I [saw [who]]]
3. [I [saw [who]]] $\xrightarrow{\text{move}}$ [who [I [saw [~~who~~]]]
4. [know]+[who [I [saw [~~who~~]]] $\xrightarrow{\text{merge}}$ [know [who [I [saw [~~who~~]]]]
5. [know [who [I [saw [~~who~~]]]]+[I] $\xrightarrow{\text{merge}}$ [I [know [who [I [saw [~~who~~]]]]]

But step 3 shows *who* being copied and deleted, revising the structure built by step 2. One response is to say that the syntax simply copies the earlier structure (perhaps only adding a link, a pointer to the embedded *who*), and then a post-syntax "spellout" process determines which copies to pronounce. This pushes the changes to completed structure out of the syntax, by invoking a "spellout" process that is sensitive to much of the same structure that syntactic operations are sensitive to. When two processes seem to be sensitive to the same structure it is a natural hunch that they are really the *same* process. Adopting this perspective instead, we could then say that the depiction

[1] Tree transducer composition, 'deforestation', is a common step for reducing program complexity (Kühnemann, 1999, Reuther, 2003, Maneth, 2004).

of the derivation 1-5 is slightly misleading: when *who* is introduced in step 1, it satisfies a requirement of the verb but is not actually placed in complement position. Rather, it is held out to be placed at the left edge of the embedded clause. This strategy for (not postponing but) eliminating a kind of structural revision is formalized in MGS (Stabler and Keenan, 2003, Frey and Gärtner, 2002, Michaelis, 2001, Harkema, 2001, Lecomte and Retoré, 1999), but MGS do not ban improper movements.

Now consider the coindexed elements in sentences like these:

He$_i$ tries [e_i to succeed]
He$_i$ laughs [before e_i eating]

These 'obligatory control' (OC) relations have enough in common with movement to suggest a uniform treatment (Hornstein, 2006, 2001, 1999, Polinsky and Potsdam, 2002, Bowers, 1973). If we generalize traditional movement so that a subject can move to another subject position even out of an adjunct as in the latter example, the rest of the phrasal construction can remain completely standard. But such movements between unconnected structures must be restricted to avoid unwanted movements, like these for example:

*John$_i$ likes t_i
*The cook they$_i$ like tried [t_i to make them]
*John$_i$ persuaded Mary [t_i to make them]
*John$_i$'s friends prefer [t_i to behave himself]

One critique of movement analyses of control wonders, if sideways movement is allowed, what rules out sideward movement from complements generally (Landau, 2003, p.477). In the present account, the status and restrictions on sideward movement will be clear: sideward movement from complements is impossible.

Another kind of problem is posed by head movements like this:

[-an]+[ustedes [habl- [español]]] → [[habl-an] [ustedes [~~habl-~~ [español]]]]

If we say *x* c-commands *y* in a tree iff a sister of *x* dominates *y*, then *habl-* does not c-command its original position. Adapting a proposal from Nunes (2001) and Hornstein (2001), in analogy to phrasal movement, we can compute this result without surgery by keeping the head *habl-* out of its projection so that is available for attachment to the appropriate affix. But the indicated assembly of the head and affix with the rest of the projection is more complicated than any of the other (merge,move) rules, looking suspiciously *ad hoc*. An alternative is to, in effect, allow the head to move before it projects its structure. This yields essentially the same result, but by allowing the head to simply move to another projection, allows the construction of the phrase and the selection of that phrase to be completely standard. But obviously this step needs to bring some analog of the traditional head movement constraint (HMC):

*be -s he have ~~be~~ -en making tortillas

Conventional movements relate source constituents with targets that c-command them. In MGs, the same effect is achieved by keeping the sources separate from the target while they wait for their final licensed positions. In this setting, the needed generalization simply allows new, ‘disconnected’ elements to be *inserted* into an expression. With this generalization of expressions, we need only one feature-checking operation, *merge*. We define ‘sideward movement grammars’ (SMMGS) in this way. To avoid overgeneralization, we impose a specifier island constraint (SpIC) and also impose a generalized ban on improper movements. Since all phrases other than the matrix clause are either complements or specifiers, SpIC allows extracted phrases to enter a derivation only through complements, though as explained below this constraint is weaker than usual because a complement can be remnant-moved to a specifier without freezing any of its moving elements.

Formal antecedents include tree adjoining grammar (Joshi and Schabes, 1997) and especially the variants proposed for scrambling (Rambow et al., 2001, Rambow, 1994, Kallmeyer, 1999), certain elaborations of pregroup grammars (Stabler, 2004a, Casadio and Lambek, 2002, Buszkowski, 2001), and the minimalist grammars (MGS) already mentioned. The derivations in these formalisms all extend and simplify complexes of possibly discontinuous constituents. But none of them enforces the ban on improper movements, and none of them defines the same class of languages as SMMGS. SMMG languages are not all MCFG definable, but they are all PMCFG-definable (Seki et al., 1991) and hence are polynomially parsable. We conjecture that all PMCFG languages are SMMG definable too.

10.2 Sideward movement grammars

Let Σ be a finite vocabulary, associated with phonetic and semantic properties. The empty sequence is ϵ. Head movement will be triggered by a morphological property that we indicate with hyphens: a preceding hyphen -s indicates that a lexical head is a suffix; a following hyphen s- indicates a prefix; and the affix s can be empty.

A set of syntactic features $\mathbb{F}$ is partitioned into 2 basic kinds: properties -$\mathbb{F}$ and requirements +$\mathbb{F}$. Properties -$\mathbb{F}$ are either persistent -f or not -$\overline{\text{f}}$. Requirements +$\mathbb{F}$: some simply require agreement +f, others trigger overt movement +$\underline{\text{f}}$, and others trigger overt movement and also leave a copy +$\underline{\underline{\text{f}}}$. As in MGS, we use the types $\mathbb{T} = \{::, :\}$ to indicate lexical and derived expressions, respectively. The *projections* $\mathbb{P} = \Sigma^* \times \mathbb{T} \times \mathbb{F}^*$. The *expressions* $\mathbb{E} = \mathbb{P} \times \wp(\mathbb{P})$. Consider, e.g., the expression (loves:-v,{Mary:-$\overline{\text{focus}}$, who:-$\overline{\text{case}}$ -wh}).

To reduce clutter, we often omit some braces and parentheses,

loves:-v, Mary:-$\overline{\text{focus}}$, who:-$\overline{\text{case}}$ -wh.

With this simpler notation, remember that the head of an expression comes first, and the order of remaining elements (if any) is irrelevant.

A *lexicon* is a finite subset of $\Sigma^* \times \{::\} \times (+\mathbb{F}^* \times -\mathbb{F}^+) \times \{\emptyset\}$ with a designated 'start' category f. A lexical item has *category* f iff its first property is -f or -$\overline{\text{f}}$. f *comp-selects* g iff there as a lexical item with category f whose first requirement is +g or +$\underline{\text{g}}$ or +$\underline{\underline{\text{g}}}$. A *cycle* is a sequence $\text{f}_0 \ldots \text{f}_n$ such that f_0 is the start category, f_{i-1} comp-selects f_i (all $0 < i \leq n$), and no feature appears twice. f *cycle-selects* g iff f precedes g in a cycle. A lexicon is *proper* iff whenever -f precedes -g in any lexical item, some lexical item containing -f has category c and some lexical item containing -g has category d, where d cycle-selects c. With this constraint on lexicons, (Proper), we can remain neutral about whether human languages have a universal, fixed clausal structure. A grammar is given by a proper lexicon, generating the structures in the closure of lexicon with respect to the fixed structure building rules. A completed structure is one containing only one syntactic feature, the start category f. The string language is the set of yields of those completed structures.

There are two structure building relations, ins and merge. The partial binary function *ins* applies to pairs of expressions $((p, S), (q, T))$ only if (i) either (q, T) is lexical or $S = \emptyset$, and (ii) match(p, q) is defined. Its value is given by ins$((p, S), (q, T)) = (p, S \cup \{q\} \cup T)$. Condition (i) is our version of SpIC, mentioned above.

The relation merge $\subset \mathbb{E} \times \mathbb{E}$ applies to (p, S) only if there is a unique $q \in S$ such that match(p, q) is defined. Then it takes as value merge$(p, S \cup \{q\}) = (r, (S - q) \cup T)$ for each match$(p, q) = (r, T)$. The uniqueness condition on application of this function is our version of the shortest move constraint (SMC).

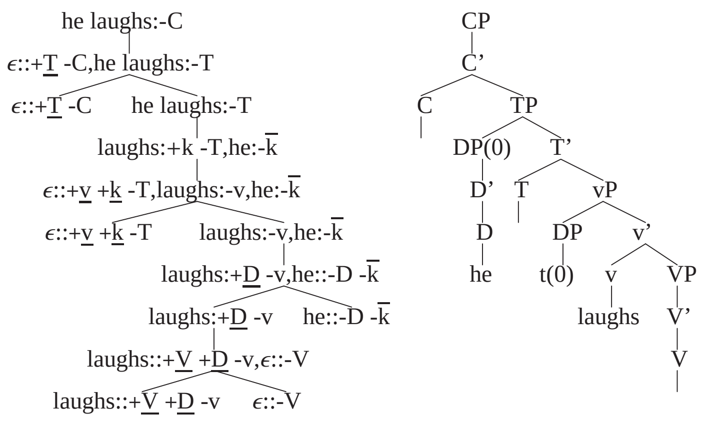
The relation match $\subset \mathbb{P} \times \mathbb{P} \times \mathbb{E}$ is given as follows, where $s, t \in \Sigma^*$ are not marked with an initial or final hyphen to trigger head movement, $\alpha, \beta, \gamma \in \mathbb{F}^*$, $\delta \in \mathbb{F}^+$, and $\cdot \in \mathbb{T}$,

Overt movement:

p	q	match(p,q)		
s::+$\underline{\text{f}}\alpha$	t·-$\overline{\text{f}}$	st:α,$\emptyset$	saturated complement	(i)
s:+$\underline{\text{f}}\alpha$	t·-$\overline{\text{f}}$	ts:α,$\emptyset$	saturated specifier	(ii)
s·+$\underline{\text{f}}\alpha$	t·-$\overline{\text{f}}\delta$	s:α,{t:δ}	moving,unsaturated projection	(iii)
s::+$\underline{\text{f}}\alpha$	t·-f	st:α,$\emptyset$	final use of -f	(iv)
s:+$\underline{\text{f}}\alpha$	t·-f	ts:α,$\emptyset$	final use of -f	(v)
s·+$\underline{\text{f}}\alpha$	t·-fδ	s:α,{t:δ}	moving,unsaturated projection	(vi)
s·+$\underline{\text{f}}\alpha$	t·-fβ	s:α,{t:-fβ}	moving with -f	(vii)
covert movement:				
s·+fα	t·-$\overline{\text{f}}\delta$	s:α,{t:δ}	check non-persistent -$\overline{\text{f}}$	(viii)
s·+fα	t·-fδ	s:α,{t:δ}	final use of -f	(ix)
s·+fα	t·-fβ	s:α,{t:-fβ}	moving with -f	(x)
copy movement:				
s::+$\underline{\underline{\text{f}}}\alpha$	t·-$\overline{\text{f}}$	st:α,$\emptyset$	saturated complement	(xi)
s:+$\underline{\underline{\text{f}}}\alpha$	t·-$\overline{\text{f}}$	ts:α,$\emptyset$	saturated specifier	(xii)
s::+$\underline{\underline{\text{f}}}\alpha$	t·-$\overline{\text{f}}\delta$	st:α,{t:δ}	moving	(xiii)
s:+$\underline{\underline{\text{f}}}\alpha$	t·-$\overline{\text{f}}\delta$	ts:α,{t:δ}	moving	(xiv)
s::+$\underline{\underline{\text{f}}}\alpha$	t·-f	st:α,$\emptyset$	final move to complement	(xv)
s:+$\underline{\underline{\text{f}}}\alpha$	t·-f	ts:α,$\emptyset$	final move to specifier	(xvi)
s::+$\underline{\underline{\text{f}}}\alpha$	t·-f	st:α,{t:-fβ}	moving with -f	(xvii)
s:+$\underline{\underline{\text{f}}}\alpha$	t·-f	ts:α,{t:-fβ}	moving with -f	(xviii)

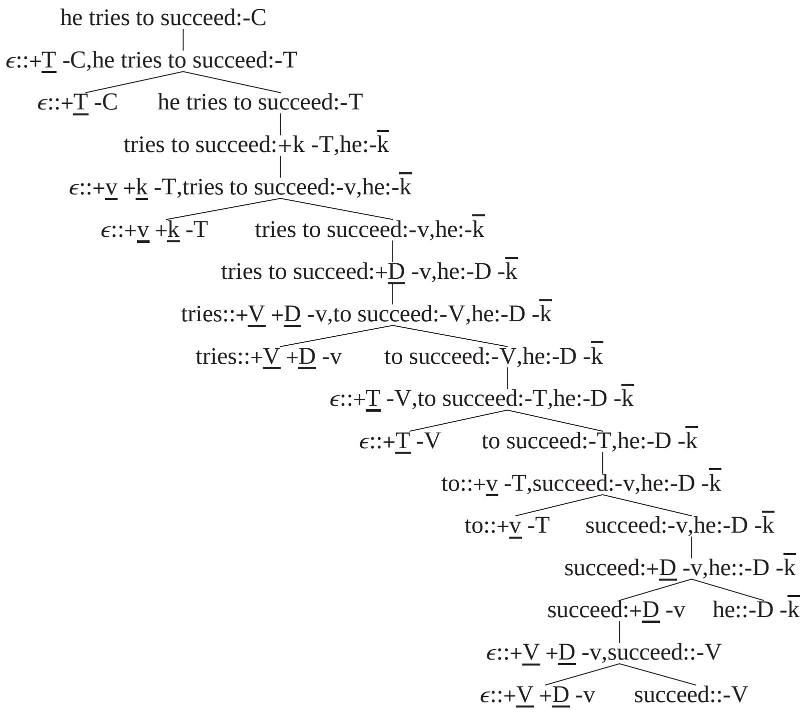
We present some examples to illustrate these mechanisms and set the stage for introducing sideward movement.

Example 1: Basics. In the derivation tree on the left, the leaves are lexical items; The binary branches represent applications of insert, and the unary branches, applications of merge.



Note that since insert applies to introduce a projection that can be merged, and the derivation greedily checks features at the earliest possible moment, there is a merge immediately above each insert step. The additional unary branches represent 'external merge' steps: these are the steps that are traditionally called 'movements'. The tree on the right shows the corresponding conventional X-bar structure. It is not difficult to translate the derivations shown here into more traditional depictions like this.[2]

Example 2: Obligatory control into a complement. One idea about obligatory control is that there is a special unpronounced pronoun PRO which, unlike other pronouns, either does not need case or else needs some special kind of case that infinitival tense can assign. But Hornstein argues that the PRO positions can be the empty positions left by movement, as in:

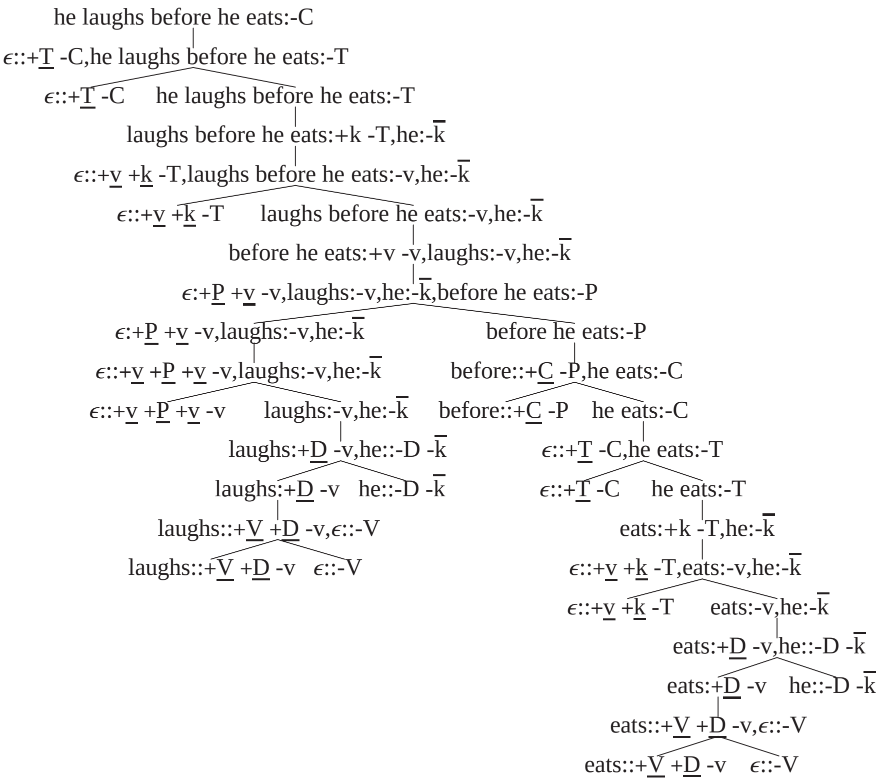


This derivation is checking the categorial D feature of [he] twice (and then checking its case feature in a higher clausal position, in conformity with Proper). Hornstein suggests that really it is θ-features getting checked twice in constructions like this. (And there have been suggestions that categorial

[2]This translation can be done automatically. See the implementations at `http://www.linguistics.ucla.edu/people/stabler/epssw.htm`.

features generally should be replaced by appropriate complexes of more basic features: θ-features etc.) For present purposes, the simple analysis above provides a suitable starting point.

Example 3: Obligatory control into an adjunct. There are many interesting questions about adjunction, but for present purposes it suffices to adopt a treatment that allows it to be category-preserving, iterable, optional, and opaque to extraction. These properties can be obtained by introducing an empty category to host the adjunct; for clausal adjuncts of noun phrases we use $\epsilon{:}{+}\underline{\mathrm{N}}{+}\underline{\mathrm{C}}{+}\underline{\mathrm{N}}{-}\mathrm{N}$, and for prepositional modifiers of v we can use: $\epsilon{::}{+}\underline{\mathrm{v}}{+}\underline{\mathrm{P}}{+}\underline{\mathrm{v}}{-}\mathrm{v}$, as in:



The fact that [before he eats] is a specifier is indicated by the non-lexical status of the selector [$\epsilon{:}{+}\underline{\mathrm{P}}\ {+}\underline{\mathrm{v}}\ {-}\mathrm{v}$,laughs:-v,he:-$\overline{\mathrm{k}}$,before he eats:-P]. Since SpIC blocks any extraction from specifiers, we do not need to separately stipulate that adjuncts are islands. So if we introduce right and left X-adjuncts of Y with lexical items of the form $\epsilon{::}{+}\underline{\mathrm{X}}{+}\underline{\mathrm{Y}}{+}\underline{\mathrm{X}}{-}\mathrm{X}$, or $\epsilon{::}{+}\underline{\mathrm{X}}{+}\underline{\mathrm{Y}}{-}\mathrm{X}$, respectively (or with any processes that yields similar structure), we get the desired properties for adjuncts: optionality, iterability, and opacity to extraction. This sets the stage for the special treatment of adjunct control.

Since the proposed treatment of adjuncts makes them opaque to extraction, while the proposed treatment of control makes it an extraction relation, we should not get control into adjuncts, but we do:

$$\text{he}_i \text{ laughs before } e_i \text{ eating}$$

Hornstein notices that a slight tweak on our mechanisms can let this kind of case through without allowing other kinds of adjunct extractions. Roughly, if we derive the modifier [before e_i eating,{he}] which wants to attach to a v, and then we derive a v that is looking for a D, we can allow [he] to 'move sideways' onto the v before inserting it into the derivation. This step can be presented in logicians' style, as the inference from the expressions above the line to the one below:

$$\frac{\text{before eating : -P}, \{\text{he : -D-}\overline{\text{k}}\} \qquad \epsilon : +\underline{\text{v}}+\underline{\text{P}}+\underline{\text{v}}\text{-v}, \emptyset \qquad \text{laughs : }+\underline{\text{D}}\text{-v}, \emptyset}{\text{laughs before eating : -v}, \{\text{he : -D-}\overline{\text{k}}\}}$$

We express this step more generally as follows. In a grammar that contains left X-adjuncts of Y, that is, it has some

$$r = \epsilon{::}+\underline{\text{X}}+\underline{\text{Y}}+\underline{\text{X}}\text{-X}$$

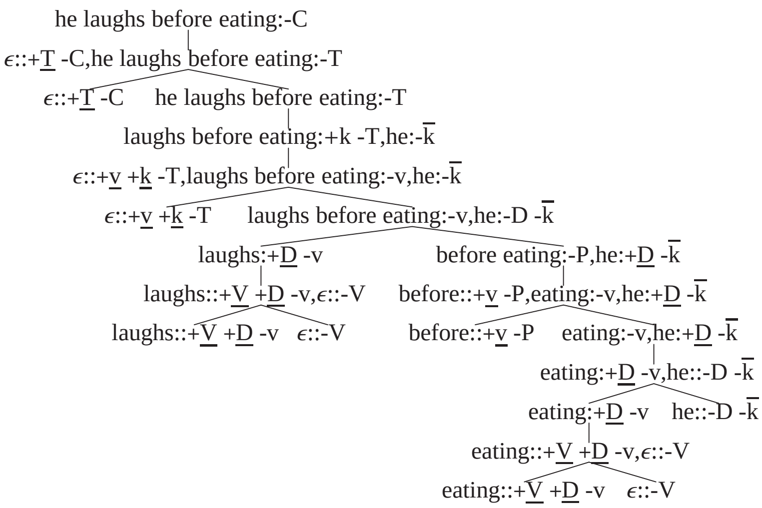
we extend the (ins) relation so that it also applies to $((p, \{a\}), (q, S))$ in the exceptional case where p and q can be chained together by r, using a as follows:

$$\begin{aligned}
&\text{match}(q, a) = (b, T),\\
&\text{match}(r, b) = (c, U),\\
&\text{match}(c, p) = (e, V), \text{ and}\\
&\text{match}(e, f) = (g, W) \text{ for } f \in U.
\end{aligned}$$

Notice that the adjoining element r is introduced in the second step to have its 3 initial features checked in sequence. In this special case, let

$$\text{ins}((p, S), (q, T)) = (g, S \cup T \cup (U - \{f\}) \cup V \cup W).$$

Control into right X-adjuncts of Y can be defined similarly, using the lexical item $\ell = \epsilon{::}+\underline{\text{X}}+\underline{\text{Y}}\text{-X}$, checking its 2 initial features in sequence. With this extension, we obtain:



Example 4: Head movement is similar to adjunct control in relating constituents that do not c-command each other, but, unlike control, we want just the phonetic parts of the heads to move while their projections are developed in their original positions. Nevertheless, there is an application of the sideward movement idea that avoids splitting all phrases kept into triples so that the head can be separate when the phrase is complete, as was done in Stabler (2001).

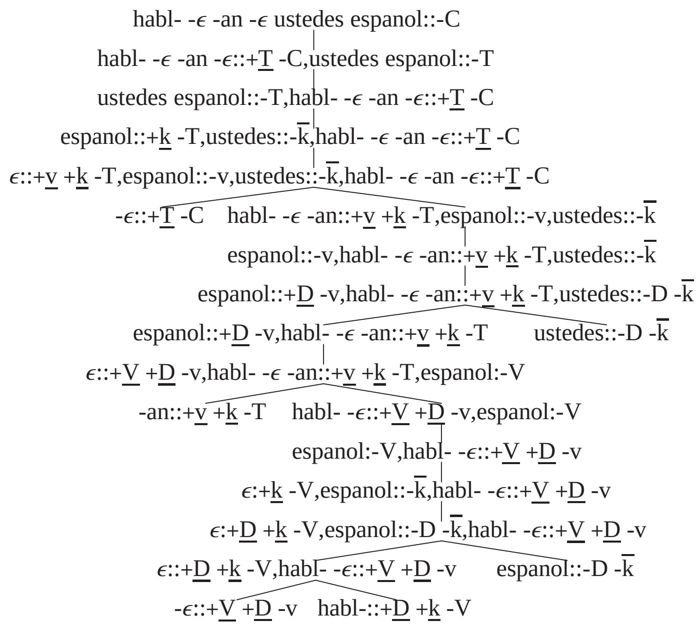
We extend match so that, when the category of -s::α is comp-selected by t::β and t-s is morphologically well-formed,

p	q	match(p,q)	
-s::α	t::β	ϵ::α,{t-s::β}	suffix left adjoins lower head
s-::α	t::β	ϵ::α,{s-t::β}	prefix right adjoins lower head

And then, when match(q, p) is defined by one of (i-xviii) we bring the adjunction up:

p	q	match(p,q)	
p	q	q,{p}	higher head promoted

With these extensions, we get derivations like the following:



No revisions of completed structure are needed, and there is no need to treat every phrase as a triple of strings.

10.3 Expressive power and recognition complexity

Previous studies have shown that head movement, though it may seem like a small thing in informal presentations, allows the definition of non-context free patterns even when there is no phrasal movement in the grammar. But the translation from MGs to MCFGs defined by Michaelis (2001) is easily adapted to show that SMMG grammars without copying all define MCFG definable languages. There are various theory-internal arguments for copying in grammar, and various ways to implement them (Stabler, 2004b). See for example Nunes (2001) and Kobele (2006) for some empirical arguments in support of rather powerful copy operations. The addition of copy features makes it easy to define non-semilinear languages like a^{2^n}, but a straightforward extension of Michaelis's translation to these cases shows that they are PMCFG-definable, and hence polynomially recognizable.

10.4 Conclusions

This paper does not attempt to resolve the controversy over whether movement analyses of obligatory control are empirically well-motivated (Landau, 2003, Boeckx and Hornstein, 2004), but provides a formalization of some parts of these ideas that can be rigorously studied.

Although SMMGS can be regarded as extending MGS, notice that they differ in a number of significant respects: (1) SMMGS extend the domain of movement just slightly to offer tightly constrained treatments of obligatory control and head movement. Future work may find ways to make these constraints more general and natural. And there are regularities in the definition of *match* that should allow a more elegant statement. (2) MGS are bound by SMC, while SMMGS also are required to respect SpIC and Proper, and future work may provide further additions. (3) To handle head movement, MGS require either extra rules for head movement (Michaelis, 2001) or else one of the approaches mentioned in the introduction. SMMGS allow head movement with a simple mechanism analogous to the sideward mechanisms used for control. (4) MGS have no copy operation, and while none of the analyses above depend on it, SMMGS allow copying. That is, we have presented a treatment of sideward movement that does not rely in any way on the copy theory of movement for its appeal. In the present setting, sideward movement is a natural option not because we already have operations on copies, but because we already have operations on moving phrases (the original phonetic materials, not copies). SMMGS are naturally extended to allow copying though, setting the stage for studying proposals about overt copying (Boeckx et al., 2005, for example) – unfortunately beyond the scope of this short report. All the mechanisms proposed here are obtained in the well-understood and feasible space of PMCFG-definable languages.

References

Boeckx, Cedric and Norbert Hornstein. 2004. Movement under control. *Linguistic Inquiry* 35(3):431–452.

Boeckx, Cedric, Norbert Hornstein, and Jairo Nunes. 2005. Overt copies in reflexive and control structures: A movement analysis. In *Workshop on New Horizons in the Grammar of Raising and Control, Harvard University*.

Bowers, John S. 1973. *Grammatical Relations*. Ph.D. thesis, Cambridge, Massachusetts, Massachusetts Institute of Technology.

Buszkowski, Wojciech. 2001. Lambek grammars based on pregroups. In P. de Groote, G. Morrill, and C. Retoré, eds., *Logical Aspects of Computational Linguistics*, Lecture Notes in Artificial Intelligence, No. 2099. NY: Springer.

Casadio, Claudia and Joachim Lambek. 2002. A tale of four grammars. *Studia Logica* 71(3):315–329.

Frey, Werner and Hans-Martin Gärtner. 2002. On the treatment of scrambling and adjunction in minimalist grammars. In *Proceedings, Formal Grammar'02*. Trento.

Harkema, Henk. 2001. *Parsing Minimalist Languages*. Ph.D. thesis, University of California, Los Angeles.

Hornstein, Norbert. 1999. Movement and control. *Linguistic Inquiry* 30:69–96.

Hornstein, Norbert. 2001. *Move! A Minimalist Theory of Construal*. Oxford: Blackwell.

Hornstein, Norbert. 2006. On control. In R. Hendriks, ed., *Contemporary Grammatical Theory*. Oxford: Blackwell. Forthcoming.

Joshi, Aravind K. and Yves Schabes. 1997. Tree-adjoining grammars. In G. Rozenberg and A. Salomaa, eds., *Handbook of Formal Languages, Volume 3: Beyond Words*, pages 69–124. NY: Springer.

Kallmeyer, Laura. 1999. *Tree Description Grammars and Underspecified Representations*. Ph.D. thesis, Universität Tübingen.

Kobele, Gregory M. 2006. Deconstructing copying: Yoruba predicate clefts and universal grammar. Presented at the LSA. http://www.linguistics.ucla.edu/people/grads/kobele/papers.htm.

Kühnemann, Armin. 1999. Comparison of deforestation techniques for functional programs and for tree transducers. In *Fuji International Symposium on Functional and Logic Programming*, pages 114–130.

Landau, Ido. 2003. Movement out of control. *Linguistic Inquiry* 34(3):471–498.

Lecomte, Alain and Christian Retoré. 1999. Towards a minimal logic for minimalist grammars. In *Proceedings, Formal Grammar'99*. Utrecht.

Maneth, Sebastian. 2004. *Models of Tree Translation*. Ph.D. thesis, Universiteit Leiden.

Michaelis, Jens. 2001. *On Formal Properties of Minimalist Grammars*. Ph.D. thesis, Universität Potsdam. *Linguistics in Potsdam 13*, Universitätsbibliothek, Potsdam, Germany.

Nunes, Jairo. 2001. Sideward movement. *Linguistic Inquiry* 32:303–344.

Polinsky, Maria and Eric Potsdam. 2002. Backward control. *Linguistic Inquiry* 33:245–282.

Rambow, Owen. 1994. *Formal and computational aspects of natural language syntax*. Ph.D. thesis, University of Pennsylvania. Computer and Information Science Technical report MS-CIS-94-52 (LINC LAB 278).

Rambow, Owen, K. Vijay-Shanker, and David Weir. 2001. D-tree substitution grammars. *Computational Linguistics* 27(1):87–121.

Reuther, Stefan. 2003. Implementing tree transducer composition for the Glasgow Haskell compiler. Diplomarbeit, Technische Universität Dresden.

Seki, Hiroyuki, Takashi Matsumura, Mamoru Fujii, and Tadao Kasami. 1991. On multiple context-free grammars. *Theoretical Computer Science* 88:191–229.

Stabler, Edward P. 2001. Recognizing head movement. In P. de Groote, G. Morrill, and C. Retoré, eds., *Logical Aspects of Computational Linguistics*, Lecture Notes in Artificial Intelligence, No. 2099, pages 254–260. NY: Springer.

Stabler, Edward P. 2004a. Tupled pregroup grammars. UCLA. Available at http://www.linguistics.ucla.edu/people/stabler/epspub.htm.

Stabler, Edward P. 2004b. Varieties of crossing dependencies: Structure dependence and mild context sensitivity. *Cognitive Science* 93(5):699–720.

Stabler, Edward P. and Edward L. Keenan. 2003. Structural similarity. *Theoretical Computer Science* 293:345–363.

11

English prepositional passives in HPSG

JESSE TSENG

Abstract

This paper discusses the treatment of English prepositional passives (also known as "pseudopassives") in HPSG. The empirical overview includes a discussion of the familiar (but unformalizable) notion of semantic cohesiveness, as well as new observations about the possibility of intervening elements between V and P. Two formal approaches to the syntactic aspects of the problem are then outlined and compared—one relying on lexical rules, the other taking advantage of HPSG's capacity to express constraints on constructions.

Keywords PSEUDOPASSIVES, PREPOSITIONS, ADJUNCTS, HPSG, LEXICAL RULES, CONSTRUCTIONS

11.1 Empirical observations

English has an exceptionally rich variety of preposition stranding phenomena, the most striking of which is the prepositional passive—the possibility of passivizing the object of a preposition instead of the direct object of a verb.

(19) a. You can rely [on David] to do get the job done.

b. David$_i$ can be <u>relied on</u> t_i to get the job done.

Here the NP *David*, initially the complement of *on*, is realized as the subject of the passive verb *relied*, leaving the preposition behind.[1]

It is often suggested that the underlined verb and preposition in this construction form a kind of "compound", an intuitive notion that is open to many

[1] I will occasionally use the symbol "*t*" to mark the "deep" position of the passive subject, in cases where there might be ambiguity. This is obviously reminiscent of NP-trace in transformational analyses, but here it should be understood only as a expository device with no theoretical strings attached.

FG-2006.
Edited by Paola Monachesi, Gerald Penn,
Giorgio Satta and Shuly Wintner.

formal interpretations. I will begin by presenting some attempts to characterize the phenomenon in semantic terms, before turning to the syntactic aspects, which will be the main focus of the rest of the paper.

11.1.1 Semantic cohesion

One semantic approach that dates back at least to the classic descriptions of Poutsma and Jespersen is the idea that the prepositional passive is possible if there is a high degree of "cohesion" between V and P. Variants of this position can be found in modern grammars (e.g., Quirk et al., 1985) and in theoretical work on preposition stranding phenomena (see Hornstein and Weinberg (1981), who propose that V and P must form a "natural predicate" or a "possible semantic word"). The most accessible indicator of semantic cohesion is the possibility of replacing the V+P sequence by a single-word synonym:

(20) David can be <u>relied on</u> ⇝ <u>trusted</u> to get the job done.

But this criterion can easily be shown to be unreliable, constituting neither a necessary nor a sufficient condition for passivizability.

It has also been observed that V+P sequences with abstract, transferred, or metaphorical meaning are more cohesive (i.e., they are more likely to allow the prepositional passive) than concrete, literal uses of the same sequence:

(21) a. An acceptable compromise was finally arrived at.

b. ??A picturesque mountain village was finally arrived at.

Similarly, semantically non-compositional combinations and idiomatic expressions can be said to be more cohesive. In these cases there would be no motivation for postulating a difference in terms of syntactic structure or function.

Other authors have attempted to analyze the prepositional passive by looking at the semantic properties of the targeted oblique NP. Bolinger (1977, 1978) proposes that this NP can become the passive subject if it refers to a strongly "affected" patient. As Riddle and Sheintuch (1983) note, no satisfactory definition is provided for this "dangerously wide" notion, and it is easy to find examples of grammatical prepositional passives where affectedness is not involved. Their own functional account (relying on the notion of "role prominence") is equally vague.[2]

Cohesion and affectedness are of course gradient properties, and they can no doubt be decomposed into more primitive, interacting factors. For example, modality, tense, and negation have all been found to influence the acceptability of the prepositional passive. Furthermore, examples that are dubious in isolation can usually be improved with an enlarged context.

[2] They themselves note that it is "impossible to offer an algorithm for determining what causes some entity or concept to be viewed as role prominent."

In this paper I make the (oversimplifying) assumption that *any* V+PP combination can give rise to a syntactically well-formed prepositional passive. The grammaticality of the resulting structure, however, is conditioned by non-syntactic restrictions that are not well enough understood to be incorporated into a formal analysis. Existing semantic accounts may be intuitively appealing but they lack a precise, empirical basis. Ultimately, we may simply have to conclude that more or less unpredictable lexical properties are the predominant factor.

11.1.2 Adjacency

A directly observable sign that V and P form a kind of "compound" in prepositional passive constructions is the fact that the insertion of adverbs and other material between V and P is generally disallowed, whereas various kinds of intervening elements are possible between V and PP in the corresponding active structure:

(22) We rely increasingly [on David] ⇝ *David is relied increasingly on.

This evidence suggests a constraint on syntactic structure and/or surface word order.[3] I will assume in this paper that intervening adverbs and PPs (modifying the verb or the VP) cannot appear in the prepositional passive. This could be formalized by introducing a word order constraint requiring V and P to be adjacent, but for various reasons this approach would be inadequate.

The specifier *right*, for instance, is (perhaps marginally) possible with some spatial and temporal Ps:[4]

(23) Mr. Cellophane may be looked right through, walked right by and never acknowledged by those who have the audacity to suppose that they cannot be looked right through.

These cases can be distinguished from (22) either structurally (*increasingly* is adjoined to V while *right* is adjoined to P) or in terms of syntactic function (*increasingly* is a modifier while *right* is a specifier). Alternatively, we could consider the facts in (23) to result from a lexical idiosyncrasy of the word *right*. But in fact other specifiers (*straight*, *clear*, etc.) can be found in similar examples, so a more general solution is called for.

Nominal elements can also separate V and P. It is well known that prepositional passives can be formed from some fixed expressions and light verb

---

[3]Note that preposition stranding by extraction is much freer in this respect (although there are restrictions, perhaps of a prosodic nature):

i We rely increasingly [on David] ⇝ David is someone that we rely increasingly on.

[4]This example is from a letter to the editor of the *Bradford Telegraph & Argus* (5 June 2003), referring to lyrics from a song: "Mr. Cellophane shoulda been my name, 'cause you can look right though me, walk right by me, and never know I'm there."

constructions containing a bare N or full NP:

(24) a. We were opened fire on, made fools of, paid attention to, taken unfair advantage of.
b. ?That product can't be made a profit from.

The commonly accepted assumption is that ordinary NP objects cannot appear between V and P, and the prepositional passive is indeed totally ungrammatical in most examples of this structure:[5]

(25) Samuel explained a complicated theorem to David. ⇝ *David was explained a complicated theorem to.

But some passived examples of the same sequence [V NP P] are surprisingly good:

(26) ?[To be whispered such dirty innuendoes about] would be enough to drive anyone crazy.

According to Bolinger (1977, 1978), the underlined direct object in this sentence functions as part of the predicate, and the passive subject (left unexpressed here) is strongly "affected" by being whispered-dirty-innuendoes-about. Another proposal by Ziv and Sheintuch (1981) requires such intervening direct objects to be "non-referential". This is a reasonable characterization of the idiomatic examples in (24), but in order to accommodate cases like (26), the authors are forced to broaden the commonly understood notion of non-referentiality considerably, and to admit that it is "not a discrete property". In the end, the acceptability of this kind of prepositional passive (and of all prepositional passives, for that matter) may depend primarily on usage and frequency effects associated with specific lexical items (or combinations of lexical itmes).

What is clear is that there can be no strict structural constraint against the presence of a direct object in the prepositional passive construction (e.g., an adjacency condition). We can also demonstrate that the ungrammaticality of the prepositional passive in cases like (25) is not due to the linear position of the direct object (between V and P). Even if the object is realized in a different position, making V and P adjacent, the prepositional passive is still totally ungrammatical:

(27) a. Samuel explained to David [a fantastically complicated theorem about the price of cheese]. (heavy NP shift)
b. *David$_i$ was explained to t_i [a fantastically complicated theorem about the price of cheese].

[5] Again, the contrast with extraction constructions is striking:

i Samuel explained a complicated theorem to David. ⇝ Who did Samuel explain a complicated theorem to?

(28) a. the theorem that Samuel explained to David / Which theorem did Samuel explain to David? (extraction)
b. *the theorem that David$_i$ was explained to t_i / *Which theorem was David$_i$ explained to t_i?

Furthermore, in cases where a direct object is possible, as in (24), there appears to be a sort of "anti-adjacency" condition on V and P. Although the direct object can be realized in various positions in the active voice, in the prepositional passive it *must* appear between V and P:

(29) a. the unfair advantage that [they took of us] / How much advantage did they take of us? (extraction)
b. *the unfair advantage that [we were taken of] / *How much advantage were we taken of?

(30) a. We could make from this product [the kinds of profits that no one has ever dreamed of] (heavy NP shift)
b. *This product$_i$ could be made from t_i [the kinds of profits that no one has ever dreamed of].

Based on these observations, I make the following assumptions for the remainder of this paper:

- The prepositional passive is syntactically compatible with the presence of a direct object.
- The direct object must be realized in its canonical position between V and P.
- The acceptability of the prepositional passive is ultimately determined by non-syntactic factors that (for now) resist formalization.

To my knowledge, only one other kind of element can intervene between V and P in the prepositional passive: when a phrasal verb is involved in this construction, its particle must appear in this position:

(31) a. This situation will simply have to be put <u>up</u> with *t*.
b. The loss in speed can be made <u>up</u> for *t* by an increase in volume.

This is unsurprising, given the strong restrictions on particle placement in English. In the active voice, the particle must be realized closest to the verb (in the absence of a direct object); this constraint continues to apply in the passive.[6]

[6] The rare examples of verbs selecting simultaneously a particle and a direct object *and* allowing the prepositional passive suggest that the relative order of the particle and the object remains the same in the active and in the passive:

(i) a. They kept <u>an eye</u> <u>out</u> for David. ⇝ ?David was kept <u>an eye</u> <u>out</u> for.
b. *They kept <u>out</u> <u>an eye</u> for David. ⇝ *David was kept <u>out</u> <u>an eye</u> for.

11.1.3 Further observations

Most of the examples given so far involve passive subjects originating in complement PPs, but it is clear that prepositional passives can also be formed from V + adjunct PP structures:

(32) a. This bed has not been slept in.

b. David always takes that seat in the corner because he hates being sat next to.

The most common sources are temporal and locative modifiers, but we also find other PPs, like instrumental *with*-phrases. Again, I will not attempt to identify or formalize the relevant semantic and lexical constraints. For the moment, I simply note that the possibility of passivizing out of adjuncts constitutes a crucial difference between the prepositional passive and the ordinary passive. [7]

We might also wonder if there is any difference between the two passives in terms of their morphological effects, given that they target different (but overlapping) sets of verbs. In particular, the prepositional passive applies to intransitive verbs like *sleep* or *go*, and to prepositional verbs like *rely*, which never undergo ordinary passivization. For verbs that do participate in both types of passivization, we might ask if two distinct morphological operations can be identified. In fact, there is no evidence for this. In every case, the same participial form is used in both constructions:

(34) a. The pilot flew the airplane under the bridge. ⇝ The airplane was <u>flown</u> *t* under the bridge. (ordinary passive)

b. The pilot flew under the bridge. ⇝ The bridge was <u>flown</u> under *t*x . (prepositional passive)

It is not the case that (say) a strong participle *flown* is used for the ordinary passive, whicle a weak form **flied* is used in the prepositional passive. Both passives require a form of the verb identical to the past participle.[8]

Finally, I briefly discuss the formation of deverbal adjectives from passive V+P sequences:

[7] NP adjuncts, for any number of reasons, cannot passivize like direct objects:

(33) The children slept three hours. ⇝ *Three hours were slept (by the children).

[8] One apparent counterexample is the following pair:

(i) a. They laid the sleeping child on the rug. ⇝ The child was <u>laid</u> *t* on the rug.

b. The child lay on the rug. ⇝ ?The rug was <u>lain/laid</u> on *t* by the child.

Here is looks as if a single verb can have a special participial form *lain* in the prepositional passive. But in fact two distinct verbs are involved in these examples: transitive *lay* (with past participle *laid*) vs intransitive *lie* (past participle ?*lain*/*laid*). This pair causes confusion and hesitation for most speakers in the past and perfect. It is safe to say, however, that no speaker merges the two into a single verb while maintaining distinct passive forms as in (34).

(35) a. our effective, relied-upon marketing strategy
b. a first novel from an as yet unheard-of author

This is sometimes taken as an additional argument for "cohesion" between V and P in the prepositional passive. For example, Hornstein and Weinberg (1981) use it to motivate the semantic notion of "possible word". It is unclear, however, what these adjectives can tell us about the passive structures they derive from, since they are evidently subject to additional constraints. Not all prepositional passives can be used to derive prenominal adjectives:

(36) a. *a sailed-under bridge, *a sat-beside grouch
b. *a taken unfair advantage of partner, *an opened fire upon enemy camp
c. *a put-up-with situation, *a made-up-for loss

Some of these examples could be improved with more context, but they all clearly have a degraded status with respect to their fully acceptable verbal counterparts. This is particularly true for the examples with an NP or particle intervening between V and P. The data suggest strongly that adjectival derivation is not a truly productive process, but is more or less lexicalized on a case by case basis. This could perhaps be accounted for with a usage-based model, but I will not pursue the idea any further here.

11.2 Implications for an HPSG analysis

11.2.1 Modularity

The normal passive construction (with the direct object NP "promoted" to subject) is standardly handled as a lexical phenomenon in HPSG, either using a lexical rule deriving the passive participle from an active base verb (Pollard and Sag, 1987), or by assuming an underspecified verbal lexeme that can be resolved to either an active or a passive form with the appropriate linking constraints (Davis and Koenig, 2000).

A number of other approaches can be imagined and technically implemented within the framework, although they have never been seriously explored. For example, new syntactic combination schemas could exceptionally realize a comps element in subject position and the subj element as a coindexed *by*-phrase. This analysis establishes a different division of labor between lexical information and syntactic operations, but it does not seem to present any advantages in return for the additional complexity it introduces.

A more radical solution would be to approximate the old transformational analysis within HPSG. A recent trend in the framework (most fully developed in Ginzburg and Sag (2001)) is the use of constructional constraints, a departure from the original emphasis (perhaps over-emphasis) on lexical descriptions as the driving force behind syntactic derivation. One characteristic of the constructional approach is a reliance on nonbranching ("head-only") syntactic rules. Such rules can potentially be used to encode arbitrarily abstract syntactic operations, from a simple change of bar level (e.g., X^0 to XP), to a coercion of one syntactic category into another (e.g., S to NP), or even in our case, the transformation of an active clause into a passive clause.

This last proposal would be soundly rejected by the linguists working in HPSG, for violating various well-motivated locality and modularity principles. In particular, a syntactic rule should not be able to refer to or arbitrarily modify the phonological, morphological, or internal syntactic structure of the constituents it manipulates. The proposed non-branching passive transformation rule would have to do all of the above. The problem is that these locality and modularity principles cannot be formally enforced in HPSG; they have the status of conceptual guidelines that responsible practitioners of the theory agree to follow by convention. Of course, this is a fundamental issue that is relevant for all grammatical frameworks, and rarely addressed. But the "all-in-one" sign-based architecture that constitutes the principal strength of HPSG, also makes it particularly easy to fall afoul of these basic principles. In the case of the passive, a constraint requiring non-branching rules to leave the PHONOLOGY and MORPHOLOGY values unchanged would be enough to invalidate the undesirable transformational analysis. But this is nothing more than an artifical stipulation, covering only a small subset of cases, and the more general theoretical question remains.

11.2.2 Adjunct analyses

For the ordinary passive construction, a strictly lexical analysis is available, because it only needs to refer to the subject and direct object, both of which are present in the lexically defined "argument structure" (encoded in the ARG-ST list). The PP adjunct data in (32), however, is problematic for a treatment of the prepositional passive as a lexical phenomenon. This is because information about the identity of eventual adjuncts is not normally available at the lexical level, at least not according to the original assumptions of HPSG. A technical work-around to this problem is possible, in the form of the DEPENDENTS list of Bouma et al. (2001). This list, whose value is defined as the lexical ARG-ST extended by zero or more (underspecified) adjuncts, was introduced in order to allow a uniformly head-driven analysis of extraction from complement and adjunct positions.

This result is made possible basically by treating some adjuncts as complements, from a syntactic point of view. This reverses the direction of selection in adjunct structures: The head now selects these adjuncts, in complete contrast to the treatment of adjuncts in Pollard and Sag (1994). This move potentially introduces significant problems for semantic composition. Levine (2003) discusses a problem involving adjuncts scoping over coordi-

$$\begin{bmatrix} \text{HEAD} & \begin{bmatrix}\text{VFORM} & \textit{base}\end{bmatrix} \\ \text{DEPS} & \left\langle \text{NP}_i, \boxed{1}\,(\text{Prt} \vee \text{NP}[\textit{canon}]), \text{PP} \right\rangle \oplus \boxed{2} \end{bmatrix} \mapsto$$

$$\begin{bmatrix} \text{HEAD} & \begin{bmatrix}\text{VFORM} & \textit{passive}\end{bmatrix} \\ \text{DEPS} & \left\langle \text{NP}_j, \boxed{1}, \text{P}\begin{bmatrix}\text{COMPS} & \langle \text{NP}_j \rangle\end{bmatrix} \right\rangle \oplus \boxed{2} \oplus \langle \text{PP}_i[\textit{by}] \rangle \end{bmatrix}$$

FIGURE 1 Prepositional Passive LR

nated structures, and argues for a return to the earlier HPSG approach, with adjuncts introduced at the appropriate places in the syntactic derivation (perhaps as empty elements, if they are extracted). Sag (2005) offers a response, requiring modifications to the proposal by Bouma et al. but maintaining the treatment of certain adjuncts as elements selected lexically by the head (and a traceless analysis of extraction).

11.2.3 Prepositional passive: lexical approach

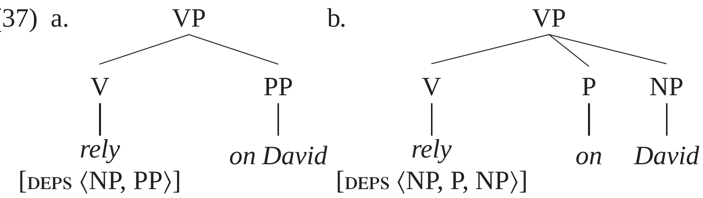
In light of this active controversy, any phenomenon involving adjuncts can be approached in two very different ways in HPSG. At first sight, the adjuncts-as-complements approach seems more appropriate for the prepositional passive, precisely because it targets complement and adjunct PPs in the same way. The lexical rule in Figure 1 takes as input a base form (active voice) verb with a PP on its DEPS list and outputs a passive participle with a DEPS specification custom-built to generate the prepositional passive: The first element on DEPS is the subject, followed optionally by a particle or a direct object.[9] The direct object, if present, is constrained to be canonical, to account for the data in (29–30) above. (Extracted and extraposed/shifted phrases have non-canonical *synsem* types.) The crucial operation in this lexical rule is the replacement of a saturated PP (complement or adjunct) in the input by a COMPS-unsaturated P in the output description. The unrealized complement of the preposition is coindexed with the passive subject NP, and the original subject is optionally realized in a *by*-phrase, as in the ordinary passive construction.

The complexity and ad hoc nature of this rule is perhaps forgivable, given the highly exceptional status of the phenomenon it models. On the other hand, the proposal fails to capture what is common to the prepositional passive and the ordinary passive. In fact, most aspects of the prepositional passive could be handled by the existing rule for the ordinary passive, which already provides a mechanism for: promoting a non-subject NP to subject position, de-

[9] This simplified formulation does not accommodate structures containing both a particle and an object (recall fn. 6).

moting the subejct NP to an optional *by*-phrase, and ensuring the appropriate morphological effects (identical for both kinds of passive, as confirmed in §11.1.3). For this to work, the NP complement of P must be made available directly on the DEPS list of the base verb (by applying argument raising, familiar from HPSG analyses of French and German non-finite constructions[10]) so it can be input to the general passive rule. But this means introducing a systematic ambiguity between PP and P, NP in the DEPS value of the active form of the verb, potentially giving rise to two structures:

(37) a. b.



The unwanted analysis (37b) should be blocked, although we need this version of the verb *rely* in order to generate the prepositional passive *was relied on*. One straightforward way to achieve this would be to add the specification *non-canonical* to the second NP element on the verb's DEPS list. This would make it impossible for it to be realized as a complement, as in (37b), but we would still have spurious ambiguity in extraction constructions (where the NP is in fact non-canonical). A more adequate solution would be to enrich the hierarchy of *synsem* subtypes to encode the syntactic function of the corresponding phrase. This would then allow us to state the appropriate constraint (e.g., "¬*comps-synsem*").[11]

This analysis of the prepositional passive is still incomplete, because the insertion of intervening modifiers between V and P must blocked; recall the discussion of example (22). The lexical operations proposed so far manipulate the DEPS list, a rather abstract level of representation that cannot be used to express constraints on surface word order. The required constraint therefore has to be formulated separately.

11.2.4 Prepositional passive: syntactic approach

A more radical treatment can be developed for the prepositional passive by combining the earlier HPSG approach to adjuncts (as unselected elements introduced in the syntax) and the more recent trend of constructional analysis.

Figure 2 sketches a special head-adjunct rule that can be used to construct the adjunct-based examples in (32). As in an ordinary head-adjunct phrase, semantic composition is handled via MOD selection. But this rule is extraor-

[10] E.g., Hinrichs and Nakazawa (1994) and Abeillé et al. (1998).

[11] This can be thought of as a very weak kind of inside-out constraint (as used in LFG, and reinterpreted for HPSG by Koenig (1999)).

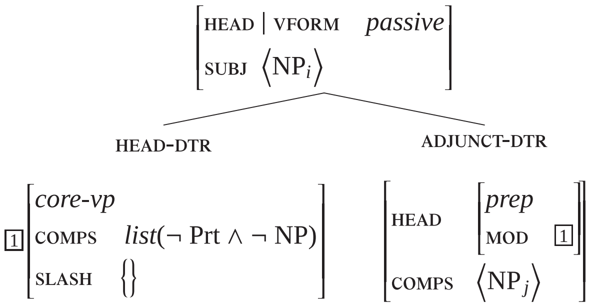


FIGURE 2 Constructional rule for adjunct prepositional passives

dinary in that it requires the adjunct to be COMPS-unsaturated, and it specifies the coindexation of the unrealized complement of P and the as-yet-unrealized subject of the resulting VP. The rule also imposes special constraints on the head daughter. The sign type *core-vp* is defined to be compatible with a bare V, or a combination of V with a particle and/or a direct object. In other words, as soon as a verb combines with a non-nominal complement or any kind of modifier, the resulting phrase is no longer a *core-vp*. This determines what can and cannot intervene between V and P in the prepositional passive, as discussed in §11.1.2 The negative constraint on the head daughter's COMPS list and the empty SLASH specification ensure that the particle and object (if any) are actually realized within the *core-vp*.[12]

A number of additional details need to be worked out; in particular, some aspects of passivization (e.g., morphological effects) must still be dealt with at the lexical level. It should also be noted that a similar special version of the head-complement rule is needed for prepositional passives involving PP complements, although it is possible to factor out the shared aspects of the two constructional rules; this is precisely the advantage of the hierarchical approach to constructions in HPSG. These preliminary observations suggest that the constructional treatment provides a more satisfactory account of the phenomenon than the lexical approach. Additional questions for further work include a comparison with the prepositional passive in Scandinavian, and a search for similar phenomena anywhere outside of the Germanic family.

References

Abeillé, Anne, Danièle Godard, and Ivan A. Sag. 1998. Two kinds of composition in French complex predicates. In E. Hinrichs, A. Kathol, and T. Nakazawa, eds.,

[12]This presupposes a return to syntactic SLASH amalgamation, as in the original HPSG Nonlocal Feature Principle.

Complex Predicates in Nonderivational Syntax, vol. 30 of *Syntax and Semantics*, pages 1–41. New York: Academic Press.

Baltin, Mark and Paul M. Postal. 1996. More on reanalysis hypotheses. *Linguistic Inquiry* 27:127–145.

Bolinger, Dwight. 1977. Transitivity and spatiality: The passive of prepositional verbs. In A. Makkai, V. B. Makkai, and L. Heilmann, eds., *Linguistics at the Crossroads*, pages 57–78. Lake Bluff, IL: Jupiter Press.

Bolinger, Dwight. 1978. Passive and transitivity again. *Forum Linguisticum* 3:25–28.

Bouma, Gosse, Rob Malouf, and Ivan A. Sag. 2001. Satisfying constraints on extraction and adjunction. *Natural Language and Linguistic Theory* 19:1–65.

Davis, Anthony and Jean-Pierre Koenig. 2000. Linking as constraints on word classes in a hierarchical lexicon. *Language* 76:56–91.

Ginzburg, Jonathan and Ivan A. Sag. 2001. *Interrogative Investigations: The Form, Meaning and Use of English Interrogatives*. Stanford, CA: CSLI Publications.

Hinrichs, Erhard and Tsuneko Nakazawa. 1994. Linearizing AUXs in German verbal complexes. In J. Nerbonne, K. Netter, and C. Pollard, eds., *German in Head-Driven Phrase Structure Grammar*, vol. 46 of *CSLI Lecture Notes*, pages 11–37. Stanford, CA: CSLI Publications.

Hornstein, Norbert and Amy Weinberg. 1981. Case theory and preposition stranding. *Linguistic Inquiry* 12:55–91.

Koenig, Jean-Pierre. 1999. Inside-out constraints and description languages for HPSG. In G. Webelhuth, J.-P. Koenig, and A. Kathol, eds., *Lexical and Constructional Aspects of Linguistic Explanation*, pages 265–279. Stanford, CA: CSLI Publications.

Levine, Robert D. 2003. Adjunct valents: cumulative scoping adverbial constructions and impossible descriptions. In J.-B. Kim and S. Wechsler, eds., *Proceedings of the 9th International HPSG Conference*, pages 209–232. Stanford, CA: CSLI Publications.

Pollard, Carl and Ivan A. Sag. 1987. *Information-Based Syntax and Semantics, Volume 1: Fundamentals*. Stanford, CA: CSLI Publications.

Pollard, Carl and Ivan A. Sag. 1994. *Head-Driven Phrase Structure Grammar*. Stanford, CA: CSLI Publications. Distributed by University of Chicago Press.

Quirk, Randolph, Sidney Greenbaum, Geoffrey Leech, and Jan Svartik. 1985. *A Comprehensive Grammar of the English Language*. London: Longman.

Riddle, Elizabeth and Gloria Sheintuch. 1983. A functional analysis of pseudo-passives. *Linguistics and Philosophy* 6:527–563.

Sag, Ivan A. 2005. Adverb extraction and coordination: a reply to Levine. In S. Müller, ed., *Proceedings of the 12th International Conference on HPSG*, pages 322–342. Stanford, CA: CSLI Publications.

Ziv, Yael and Gloria Sheintuch. 1981. Passives of obliques over direct objects. *Lingua* 54:1–17.

12

Linearization of Affine Abstract Categorial Grammars

Ryo Yoshinaka

Abstract

The abstract categorial grammar (ACG) is a grammar formalism based on linear lambda calculus. It is natural to ask how the expressive power of ACGs increases when we relax the linearity constraint on the formalism. This paper introduces the notion of affine ACGs by extending the definition of original ACGs, and presents a procedure for converting a given affine ACG into a linear ACG whose language is exactly the set of linear λ-terms generated by the original affine ACG.

Keywords Abstract Categorial Grammars, Generative Capacity, Lambda Calculus, Context-Free Tree Grammars, Linear Context-Free Rewriting Systems, Multiple Context-Free Grammars

12.1 Introduction

De Groote (2001) has introduced *abstract categorial grammars (ACGs)*, in which both *lexical entries* of the grammar as well as *grammatical combinations* of them are represented by simply typed linear λ-terms. While the linearity constraint on grammatical combinations is thought to be reasonable, admitting non-linear λ-terms as lexical entries may allow ACGs to describe linguistic phenomena in a more natural and concise fashion.

On the other hand, de Groote and Pogodalla (2003, 2004) have shown that a variety of context-free formalisms, namely, context-free grammars, linear[1]

[1] This paper lets the term "linearity" mean non-duplication and non-deletion. Thus "linear CFTGs" means non-duplicating non-deleting CFTGs here, though usually "linear CFTGs" means non-duplicating CFTGs.

FG-2006.
Edited by Paola Monachesi, Gerald Penn,
Giorgio Satta and Shuly Wintner.

context-free tree grammars (linear CFTGs)[2] and linear context-free rewriting systems (LCFRSs), is encoded by ACGs in straightforward ways. In this sense, ACGs can be thought of as a generalization of those grammar formalisms. The linearity constraint in those formalisms matches that of the ACG formalism.

Concerning those grammar formalisms, it is known that the expressive power does not change when the linearity constraint is relaxed to just non-duplication, allowing deleting operations. Seki et al. (1991) have shown the equivalence between LCFRSs and multiple context-free grammars (MCFGs), which correspond to the relaxed version of LCFRSs that may have deleting operations. Fujiyoshi (2005) has established the equivalence between linear monadic CFTGs and non-duplicating monadic CFTGs. Fisher's result (Fisher, 1968a,b) is rather general. He has shown that the string IO-languages generated by general CFTGs coincide with the string IO-languages generated by non-deleting CFTGs.

Along this line, extending the definition of usual linear ACGs, this paper introduces *affine ACGs*, which have BCKλ-terms as their lexical entries, and compares the generative power of linear ACGs and affine ACGs. We present a procedure for converting a given affine ACG into a linear ACG whose language is exactly the set of the linear λ-terms generated by the original ACG. Therefore, affine ACGs are not essentially more expressive than linear ACGs, since strings and trees are usually represented with linear λ-terms.

As linear ACGs encode linear CFTGs and LCFRSs, affine ACGs encode non-duplicating CFTGs and MCFGs in straightforward ways. For such affine ACGs, our linearization method constructs linear ACGs which have the form corresponding to linear CFTGs or LCFRSs. Thus, our result is a generalization of the results we have mentioned above with the exception of Fisher's, which covers CFTGs involving duplication.

12.2 Preliminaries

12.2.1 Lambda-Terms

Let $\mathscr{A}$ be a finite non-empty set of *atomic types*. The set $\mathcal{T}(\mathscr{A})$ of *types* built on $\mathscr{A}$ is defined as the smallest superset of $\mathscr{A}$ such that

- if $\alpha, \beta \in \mathcal{T}(\mathscr{A})$, then $(\alpha \to \beta) \in \mathcal{T}(\mathscr{A})$.

The *order* of a type is given by the function $\mathrm{ord} : \mathcal{T}(\mathscr{A}) \to \mathbb{N}$,

- $\mathrm{ord}(p) = 1$ for all $p \in \mathscr{A}$,
- $\mathrm{ord}((\alpha \to \beta)) = \max\{\mathrm{ord}(\alpha) + 1, \mathrm{ord}(\beta)\}$.

[2]See also Kanazawa and Yoshinaka (2005) for complete proof of encodability of linear CFTGs by ACGs.

A *higher-order signature* Σ is a triple $\langle \mathscr{A}, \mathscr{C}, \tau \rangle$ where $\mathscr{A}$ is a finite non-empty set of atomic types, $\mathscr{C}$ is a finite set of constants, and τ is a function from $\mathscr{C}$ to $\mathcal{T}(\mathscr{A})$. The *order* of the higher-order signature is defined as $\mathrm{ord}(\Sigma) = \max\{\, \mathrm{ord}(\tau(\mathsf{a})) \mid \mathsf{a} \in \mathscr{C}\,\}$.

Let $\mathscr{X}$ be a countably infinite set of *variables*. The set $\Lambda(\Sigma)$ of λ-*terms* (*terms* for short) built upon Σ and the type $\hat{\tau}(M)$ of a term $M \in \Lambda(\Sigma)$ are defined inductively as follows:

- For every $\mathsf{a} \in \mathscr{C}$, $\mathsf{a} \in \Lambda(\Sigma)$ and $\hat{\tau}(\mathsf{a}) = \tau(\mathsf{a})$.
- For every $x \in \mathscr{X}$ and $\alpha \in \mathcal{T}(\mathscr{A})$, $x^{\alpha} \in \Lambda(\Sigma)$ and $\hat{\tau}(x^{\alpha}) = \alpha$.
- For $M, N \in \Lambda(\Sigma)$, if $\hat{\tau}(M) = (\alpha \to \beta)$, $\hat{\tau}(N) = \alpha$, then $(MN) \in \Lambda(\Sigma)$ and $\hat{\tau}((MN)) = \beta$.
- For $x \in \mathscr{X}$, $\alpha \in \mathcal{T}(\mathscr{A})$ and $M \in \Lambda(\Sigma)$, $(\lambda x^{\alpha}.M) \in \Lambda(\Sigma)$ and $\hat{\tau}((\lambda x^{\alpha}.M)) = (\alpha \to \hat{\tau}(M))$.

For convenience, we simply write τ instead of $\hat{\tau}$ and often omit the superscript on a variable if its type is clear from the context. The notions of free variables, closed terms, β-normal form, $\beta\eta$-normal form, are defined as usual (see Hindley (1997) for instance). A term M is a *combinator* iff M is closed and M contains no constants. A term M is said to be *affine* if any variable occurs free at most once in every subterm of M. An affine term is said to be *linear* if every λ-abstraction binds exactly one occurrence of a variable. The sets of affine and linear terms are respectively denoted by $\Lambda^{\mathrm{aff}}(\Sigma)$ and $\Lambda^{\mathrm{lin}}(\Sigma)$. As usual, let $\twoheadrightarrow_{\beta}$, $=_{\beta}$, $=_{\beta\eta}$, $\equiv$ denote β-reduction, β-equality, $\beta\eta$-equality, and α-equivalence respectively. $|M|_{\beta}$ and $|M|_{\beta\eta}$ respectively represent the β-normal form and $\beta\eta$-normal form. We use upper case italic letters $M, N, P, \ldots$ for terms, late lower case italic letters $x, y, z, \ldots$ for variables, middle lower case italic letters $o, p, \ldots$ for atomic types, Greek letters $\alpha, \beta, \ldots$ for types, sanserif $\mathsf{a}, \mathsf{A}, \ldots$ for constants. We write $\alpha \to \beta \to \gamma \to \delta$ for $(\alpha \to (\beta \to (\gamma \to \delta)))$, $\alpha^3 \to \delta$ for $\alpha \to \alpha \to \alpha \to \delta$, $MNPQ$ for $(((MN)P)Q)$, $\lambda xyz.M$ for $(\lambda x.(\lambda y.(\lambda z.M)))$, and so on.

12.2.2 Abstract Categorial Grammars

Definition 12 For two sets of atomic types $\mathscr{A}_0$ and $\mathscr{A}_1$, a *type substitution* σ is a mapping from $\mathscr{A}_0$ to $\mathcal{T}(\mathscr{A}_1)$, which can be extended homomorphically as

$$\sigma(\alpha \to \beta) = \sigma(\alpha) \to \sigma(\beta).$$

For two higher-order signatures Σ_0 and Σ_1, a *term substitution* θ is a mapping from $\mathscr{C}_0$ to $\Lambda(\Sigma_1)$ such that $\theta(\mathsf{a})$ is closed for all $\mathsf{a} \in \mathscr{C}_0$. A term substitution θ is *linear* iff $\theta(\mathsf{a})$ is linear for all $\mathsf{a} \in \mathscr{C}_0$. For two higher-order signatures Σ_0 and Σ_1, we say that a type substitution $\sigma : \mathscr{A}_0 \to \mathcal{T}(\mathscr{A}_1)$ and a term substitution $\theta : \mathscr{C}_0 \to \Lambda(\Sigma_1)$ are *compatible* iff $\sigma(\tau_0(\mathsf{a})) = \tau_1(\theta(\mathsf{a}))$ holds for all $\mathsf{a} \in \mathscr{C}_0$. A *lexicon* from Σ_0 to Σ_1 is a compatible pair of a type substitution and a term substitution. A lexicon $\mathscr{L} = \langle \sigma, \theta \rangle$ is *linear* iff θ is linear. For a lexicon $\mathscr{L} = \langle \sigma, \theta \rangle$, we define $\hat{\theta}$ as the homomorphic extension of θ such that $\hat{\theta}(x^{\alpha}) = x^{\sigma(\alpha)}$. Indeed, $\hat{\theta}(M)$ is always a well-typed λ-term if so is M; if M has type α, then $\hat{\theta}(M)$ has type $\sigma(\alpha)$.

Hereafter we identify a lexicon $\mathscr{L} = \langle \sigma, \theta \rangle$ with the functions σ and $\hat{\theta}$. A lexicon $\mathscr{L}$ is *n-th order* if $\mathrm{ord}(\mathscr{L}) = \max\{ \mathrm{ord}(\sigma(p)) \mid p \in \mathscr{A}_0 \} \leq n$.

Definition 13 An *abstract categorial grammar (ACG)* is a quadruple $\mathscr{G} = \langle \Sigma_0, \Sigma_1, \mathscr{L}, s \rangle$, where

- Σ_0 is a higher-order signature, called the *abstract vocabulary*,
- Σ_1 is a higher-order signature, called the *object vocabulary*,
- $\mathscr{L}$ is a linear lexicon from Σ_0 to Σ_1,
- $s \in \mathscr{A}_0$ is called the *distinguished type*.

We sometimes call the triple $\langle \mathsf{a}, \tau_0(\mathsf{a}), \mathscr{L}(\mathsf{a}) \rangle$ for $\mathsf{a} \in \mathscr{C}_0$ a *lexical entry*, and specify an ACG by giving the set of lexical entries and the distinguished type.

Definition 14 An ACG $\mathscr{G} = \langle \Sigma_0, \Sigma_1, \mathscr{L}, s \rangle$ generates two languages, the *abstract language* $\mathcal{A}(\mathscr{G})$ and the *object language* $\mathcal{O}(\mathscr{G})$, defined as

$$\mathcal{A}(\mathscr{G}) = \{ M \mid M \in \Lambda^{\mathrm{lin}}(\Sigma_0) \text{ is a closed } \beta\eta\text{-normal term of type } s \},$$
$$\mathcal{O}(\mathscr{G}) = \{ |\mathscr{L}(M)|_{\beta\eta} \mid M \in \mathcal{A}(\mathscr{G}) \}.$$

The abstract language can be thought of as a set of abstract grammatical structures, and the object language is regarded as the set of concrete forms obtained from these abstract structures and the lexicon. Thus, we simply say the language generated by an ACG for its object language. The term *abstract categorial languages (ACLs)* means the object languages of ACGs.

Though de Groote's original definition of an ACG requires the lexicon to be linear, this paper allows the lexicon to be non-linear. We call an ACG whose lexicon is affine *affine ACG*, and denote the class of affine ACGs by $\mathbf{G}^{\mathrm{aff}}$. We then distinguish affine ACGs whose lexicons are linear, i.e., original ACGs, by calling them *linear ACGs* and let $\mathbf{G}^{\mathrm{lin}}$ denote the class of linear ACGs. Note that the abstract language always consists of *linear* terms, though an ACG is not necessarily linear. For each $\mathbf{G}^{*} \in \{\mathbf{G}^{\mathrm{lin}}, \mathbf{G}^{\mathrm{aff}}\}$, $\mathbf{G}^{*}(m, n)$ denotes the subclass of ACGs belonging to $\mathbf{G}^{*}$ such that the order of the abstract vocabulary is at most m and the order of the lexicon is at most n. An ACG is *m-th order* if it belongs to $\mathbf{G}^{*}(m, n)$ for some n.

Example 1 Let $\mathit{str} = o \to o$ and $M + N$ be an abbreviation of $\lambda z^{o}.M(Nz)$ if the types of M and N are str. Let us consider the affine ACG $\mathscr{G} = \langle \Sigma_0, \Sigma_1, \mathscr{L}, s \rangle$

with the following lexical entries:

$\mathsf{x} \in \mathscr{C}_0$	$\tau_0(\mathsf{x})$	$\mathscr{L}(\mathsf{x})$
C	n	$\lambda v.v/\mathsf{cat}//\mathsf{cats}/$
M	n	$\lambda v.v/\mathsf{mouse}//\mathsf{mice}/$
J	np	$\lambda y.y/\mathsf{John}/P_1$
R	$np \to s$	$\lambda x.x(\lambda uv.u + v/\mathsf{runs}//\mathsf{run}/)$
E	$np^2 \to s$	$\lambda x_1x_2.x_2(\lambda uv.u + v/\mathsf{eats}//\mathsf{eat}/) + x_1(\lambda uv.u)$
A	$n \to np$	$\lambda zy.y(/\mathsf{a}/ + zP_1)P_1$
L	$n \to np$	$\lambda zy.y(/\mathsf{all}/ + zP_2)P_2$

where each $/\mathsf{xxx}/$ is a constant of type str, P_i denotes $\lambda u_1^{\mathit{str}} u_2^{\mathit{str}}.u_i$, $\mathscr{L}(n) = (\mathit{str}^2 \to \mathit{str}) \to \mathit{str}$, $\mathscr{L}(np) = (\mathit{str} \to (\mathit{str}^2 \to \mathit{str}) \to \mathit{str}) \to \mathit{str}$, $\mathscr{L}(s) = \mathit{str}$. The object language $O(\mathscr{G})$ consists of terms representing some English sentences such as *John runs*, *all mice run*, *all cats eat a mouse*, and so on.

12.3 Linearization of Affine ACGs

While linear ACGs can generate languages consisting of linear terms only, affine ACGs can generate languages containing non-linear terms. Therefore, affine ACGs define a strictly richer class of languages than linear ACGs. However, since terms representing strings or trees are linear[3], affine terms in the object languages are not very interesting. This paper shows that for every $\mathscr{G} \in \mathbf{G}^{\mathrm{aff}}(m, n)$, we can construct $\mathscr{G}' \in \mathbf{G}^{\mathrm{lin}}(m, \max\{2, n\})$ such that

$$O(\mathscr{G}') = \{\, P \in O(\mathscr{G}) \mid P \text{ is linear} \,\} \tag{12.8}$$

Moreover, in case of $m = 2$, we can find $\mathscr{G}' \in \mathbf{G}^{\mathrm{lin}}(2, n)$ satisfying the equation (12.8). Therefore extending the definition of an ACG to allow lexical entries affine does not enrich the expressive power of ACGs in an essential way. Before proceeding with our construction, we mention a partially stronger result on the special case of this problem on string-generating second-order ACGs, obtained from Salvati's work (Salvati, 2006). He presents an algorithm that converts a linear ACG $\mathscr{G} \in \mathbf{G}^{\mathrm{lin}}(2, n)$ generating a string language into an equivalent LCFRS (via a deterministic tree-walking transducer). Even if an input is an affine ACG $\mathscr{G} \in \mathbf{G}^{\mathrm{aff}}(2, n)$, his algorithm still outputs an equivalent LCFRS. Since every LCFRS is encodable by a linear ACG belonging to $\mathbf{G}^{\mathrm{lin}}(2, 4)$ (de Groote and Pogodalla, 2003, 2004), therefore this entails the following corollary.

[3]A string $\mathsf{a}_1 \ldots \mathsf{a}_n$ on an alphabet V is represented by $\lambda z^o.\mathsf{a}_1(\ldots(\mathsf{a}_n z)\ldots) \in \Lambda^{\mathrm{lin}}(\Sigma_V)$ where $\Sigma_V = \langle\{o\}, V, \tau_V\rangle$ with $\tau_V(\mathsf{a}) = \mathit{str}$ for all $\mathsf{a} \in V$ as in Example 1. Trees are constructed on some ranked alphabet. A ranked alphabet $\langle F, \rho\rangle$, where F is an alphabet and ρ is a rank assignment on F, can be identified with a higher-order signature $\Sigma_{\langle F,\rho\rangle} = \langle\{o\}, F, \tau_\rho\rangle$ such that $\tau_\rho(\mathsf{a}) = o^k \to o$ if $\rho(\mathsf{a}) = k$ for all $\mathsf{a} \in F$, and a tree is identified with a variable-free (thus linear) term of the atomic type o in the obvious way.

Corollary 26 *For every string-generating affine ACG* $\mathscr{G} \in \mathbf{G}^{\mathrm{aff}}(2,n)$, *there is a linear ACG* $\mathscr{G}' \in \mathbf{G}^{\mathrm{lin}}(2,4)$ *such that* $\mathcal{O}(\mathscr{G}') = \mathcal{O}(\mathscr{G})$.

12.3.1 Basic Idea

We explain our basic idea for the linearization method for affine ACGs through a small example. Let us consider the affine ACG $\mathscr{G}$ consisting of the following lexical entries:

$\mathsf{x} \in \mathscr{C}_0$	$\tau_0(\mathsf{x})$	$\mathscr{L}(\mathsf{x})$
A	$p \to s$	$\lambda w^{o^2 \to o}.w\mathsf{a}^o\mathsf{b}^o$
B	p	$\lambda x^o y^o.x$

where $\mathscr{L}(s) = o$ and $\mathscr{L}(p) = o^2 \to o$. Corresponding to $\mathsf{AB} \in \mathcal{A}(\mathscr{G})$, we have $\mathsf{a} \in \mathcal{O}(\mathscr{G})$ by

$$\mathscr{L}(\mathsf{AB}) \equiv (\lambda w^{o\to o\to o}.w\mathsf{a}^o\mathsf{b}^o)(\lambda x^o y^o.x) \to_\beta (\lambda x^o y^o.x)\mathsf{a}^o\mathsf{b}^o \twoheadrightarrow_\beta \mathsf{a}^o. \tag{12.9}$$

The occurrences of vacuous λ-abstraction λy^o causes the deletion of b in (12.9). Such deleting operation is what we want to eliminate in order to linearize the affine ACG $\mathscr{G}$. Let us retype λy^o with $\lambda y^{\overline{o}}$ and replace b^o with $\overline{\mathsf{b}}^{\overline{o}}$ to indicate that they should be eliminated. Then (12.9) is decorated by bars as

$$(\lambda w^{o\to\overline{o}\to o}.w\mathsf{a}^o\overline{\mathsf{b}}^{\overline{o}})(\lambda x^o y^{\overline{o}}.x) \to_\beta (\lambda x^o y^{\overline{o}}.x)\mathsf{a}^o\overline{\mathsf{b}}^{\overline{o}} \twoheadrightarrow_\beta \mathsf{a}^o, \tag{12.10}$$

where we retype $w^{o\to o\to o}$ with $w^{o\to\overline{o}\to o}$, so that the whole term is well-typed. In our setting, when a term has a barred type, it means that the term should be erased during β-reduction steps, and vice versa. By eliminating those barred terms and types from (12.10), we get

$$(\lambda w^{o\to o}.w\mathsf{a}^o)(\lambda x^o.x) \to_\beta (\lambda x^o.x)\mathsf{a}^o \to_\beta \mathsf{a}^o, \tag{12.11}$$

which solely consists of linear terms. Hence, the linear ACG $\mathscr{G}'$ with the following lexical entries generates the same language as the original ACG $\mathscr{G}$.

$\mathsf{x} \in \mathscr{C}'_0$	$\tau'_0(\mathsf{x})$	$\mathscr{L}'(\mathsf{x})$
A′	$[p, o \to \overline{o} \to o] \to [s, o]$	$\lambda w^{o\to o}.w\mathsf{a}^o$
B′	$[p, o \to \overline{o} \to o]$	$\lambda x^o.x$

where $[p, o \to \overline{o} \to o]$ and $[s, o]$ are new atomic types that are mapped to $o \to o$ and o, respectively, and $[s, o]$ is the distinguished type. We have $\mathscr{L}(\mathsf{AB}) = \mathscr{L}'(\mathsf{A}'\mathsf{B}')$. The term $\lambda w^{o\to\overline{o}\to o}.w\mathsf{a}^o\overline{\mathsf{b}}^{\overline{o}}$, which is led to $\mathscr{L}'(\mathsf{A}')$, is just one possible bar-decoration for $\mathscr{L}(\mathsf{A})$. For instance, $\lambda w^{\overline{o}\to o\to o}.w\overline{\mathsf{a}}^{\overline{o}}\mathsf{b}^o$ and $\lambda w^{o\to o\to o}.w\mathsf{a}^o\mathsf{b}^o$ are also possible. Bars appearing in $\lambda w^{\overline{o}\to o\to o}.w\overline{\mathsf{a}}^{\overline{o}}\mathsf{b}^o$ predict that the subterm $\overline{\mathsf{a}}$ will be erased, and $\lambda w^{o\to o\to o}.w\mathsf{a}^o\mathsf{b}^o$ predicts that no subterm of it will disappear. Our linearization method also produces lexical entries corresponding to those bar-decorations.

12.3.2 Formal Definition

We first give a formal definition of the set of possible bar-decorations on a type and a term. Hereafter, we fix a given affine ACG $\mathscr{G} = \langle \Sigma_0, \Sigma_1, \mathscr{L}, s\rangle$. Define $\overline{\Sigma_1} = \langle \overline{\mathscr{A}_1}, \overline{\mathscr{C}_1}, \overline{\tau_1}\rangle$ by

$$\overline{\mathscr{A}_1} = \{\, \overline{p} \mid p \in \mathscr{A}_1 \,\},\ \overline{\mathscr{C}_1} = \{\, \overline{\mathsf{c}} \mid \mathsf{c} \in \mathscr{C}_1 \,\},\ \overline{\tau_1} = \{\, \overline{\mathsf{c}} \to \overline{\tau_1(\mathsf{c})} \mid \mathsf{c} \in \mathscr{C}_1 \,\},$$

where $\overline{\alpha \to \beta} = \overline{\alpha} \to \overline{\beta}$. Let $\Sigma_1' = \langle \mathscr{A}_1', \mathscr{C}_1', \tau_1' \rangle = \langle \mathscr{A}_1 \cup \overline{\mathscr{A}_1}, \mathscr{C}_1 \cup \overline{\mathscr{C}_1}, \tau_1 \cup \overline{\tau_1}\rangle$. Here, we have the simple lexicon $\widetilde{\cdot}$ from Σ_1' to Σ_1 defined as

$$\widetilde{\overline{p}} = \widetilde{p} = p \text{ for } p \in \mathscr{A}_1, \text{ and } \widetilde{\overline{\mathsf{c}}} \equiv \widetilde{\mathsf{c}} \equiv \mathsf{c} \text{ for } \mathsf{c} \in \mathscr{C}_1.$$

The set $\widehat{\mathcal{T}}(\mathscr{A}_1)$ of possible bar-decorations on types is defined by

$$\widehat{\mathcal{T}}(\mathscr{A}_1) = \{\, \alpha \in \mathcal{T}(\mathscr{A}_1') \mid \text{if } \beta_1 \to \cdots \to \beta_n \to \overline{p} \text{ is a subtype of } \alpha \text{ for some } \overline{p} \in \overline{\mathscr{A}_1}, \text{ then } \beta_1, \ldots, \beta_n \in \mathcal{T}(\overline{\Sigma_1}) \,\}$$

Actually, terms in $\Lambda^{\mathrm{aff}}(\Sigma_1')$ that we are concerned with have types in $\widehat{\mathcal{T}}(\mathscr{A}_1)$. The reason why we ignore types in $\mathcal{T}(\mathscr{A}_1') - \widehat{\mathcal{T}}(\mathscr{A}_1)$ is that if a term is bound to be erased, then so is every subterm of it. For instance, if a variable x has type $o \to \overline{o} \notin \widehat{\mathcal{T}}(\{o\})$, then the term $x^{o\to\overline{o}}y^o$ has type $\overline{o}$, which, in our setting, means that it should disappear. But if $x^{o\to\overline{o}}y^o$ disappears, so does y^o, which, therefore, should have type $\overline{o}$ to be consistent with our definition.

The set $\widehat{\Lambda}^{\mathrm{aff}}(\Sigma_1)$ of possible bar-decorations on terms is the subset of $\Lambda^{\mathrm{aff}}(\Sigma_1')$ such that $Q \in \widehat{\Lambda}^{\mathrm{aff}}(\Sigma_1)$ iff

- every variable appearing in Q has a type in $\widehat{\mathcal{T}}(\mathscr{A}_1)$, and
- if $\lambda x^\alpha.Q'$ is a subterm of Q and x^α does not occur free in Q', then $\alpha \in \mathcal{T}(\overline{\mathscr{A}_1})$.

We are not concerned with terms in $\Lambda^{\mathrm{aff}}(\Sigma_1') - \widehat{\Lambda}^{\mathrm{aff}}(\Sigma_1)$.

The following properties are easily seen:

- If $Q \in \widehat{\Lambda}^{\mathrm{aff}}(\Sigma_1)$, then $\tau_1'(Q) \in \widehat{\mathcal{T}}(\mathscr{A}_1)$,
- If $\tau_1'(Q) \in \mathcal{T}(\overline{\mathscr{A}_1})$ for $Q \in \widehat{\Lambda}^{\mathrm{aff}}(\Sigma_1)$, every subterm of Q is in $\Lambda^{\mathrm{aff}}(\overline{\Sigma_1})$,
- If $Q \in \widehat{\Lambda}^{\mathrm{aff}}(\Sigma_1)$ and $Q \twoheadrightarrow_\beta Q'$, then $Q' \in \widehat{\Lambda}^{\mathrm{aff}}(\Sigma_1)$.

For each $\alpha \in \mathcal{T}(\mathscr{A}_1)$ and $P \in \Lambda^{\mathrm{aff}}(\Sigma_1)$, Π gives the set of possible bar-decorations on them:

$$\Pi(\alpha) = \{\, \beta \in \widehat{\mathcal{T}}(\mathscr{A}_1) \mid \widetilde{\beta} = \alpha \,\},$$
$$\Pi(P) = \{\, Q \in \widehat{\Lambda}^{\mathrm{aff}}(\Sigma_1) \mid \widetilde{Q} \equiv P \,\}.$$

In other words, Π and $\widetilde{\cdot}$ are inverse of each other, if we disregard types in $\mathcal{T}(\mathscr{A}_1') - \widehat{\mathcal{T}}(\mathscr{A}_1)$ and terms in $\Lambda^{\mathrm{aff}}(\Sigma_1') - \widehat{\Lambda}^{\mathrm{aff}}(\Sigma_1)$.

Secondly, we eliminate barred subtypes from $\alpha \in \widehat{\mathcal{T}}(\mathscr{A}_1) - \mathcal{T}(\overline{\mathscr{A}_1})$ and barred subterms from $Q \in \widehat{\Lambda}^{\mathrm{aff}}(\Sigma_1) - \Lambda^{\mathrm{aff}}(\overline{\Sigma_1})$. Let us define $(\alpha)^\dagger$ and $(Q)^\dagger$ as follows:

$$
\begin{aligned}
(p)^\dagger &= p \quad \text{for } p \in \mathscr{A}_1,\\
(\alpha \to \beta)^\dagger &= \begin{cases} (\alpha)^\dagger \to (\beta)^\dagger & \text{if } \alpha \notin \mathcal{T}(\overline{\mathscr{A}_1}),\\ (\beta)^\dagger & \text{if } \alpha \in \mathcal{T}(\overline{\mathscr{A}_1}),\end{cases}\\
(x^\alpha)^\dagger &\equiv x^{(\alpha)^\dagger},\\
(\mathsf{c})^\dagger &\equiv \mathsf{c} \quad \text{for } \mathsf{c} \in \mathscr{C}_1,\\
(\lambda x^\alpha.Q)^\dagger &\equiv \begin{cases} \lambda x^{(\alpha)^\dagger}.(Q)^\dagger & \text{if } \alpha \notin \mathcal{T}(\overline{\mathscr{A}_1}),\\ (Q)^\dagger & \text{if } \alpha \in \mathcal{T}(\overline{\mathscr{A}_1}),\end{cases}\\
(Q_1 Q_2)^\dagger &\equiv \begin{cases} (Q_1)^\dagger (Q_2)^\dagger & \text{if } \tau_1'(Q_2) \notin \mathcal{T}(\overline{\mathscr{A}_1}),\\ (Q_1)^\dagger & \text{if } \tau_1'(Q_2) \in \mathcal{T}(\overline{\mathscr{A}_1}).\end{cases}
\end{aligned}
$$

The following properties are easily seen ($\alpha \in \widehat{\mathcal{T}}(\mathscr{A}_1) - \mathcal{T}(\overline{\mathscr{A}_1})$ and $Q, Q' \in \widehat{\Lambda}^{\mathrm{aff}}(\Sigma_1) - \Lambda^{\mathrm{aff}}(\overline{\Sigma_1})$):

- $(\alpha)^\dagger \in \mathcal{T}(\mathscr{A}_1)$ and $(Q)^\dagger \in \Lambda^{\mathrm{lin}}(\Sigma_1)$,
- $\tau_1((Q)^\dagger) = (\tau_1'(Q))^\dagger$,
- If Q is β-normal, then so is $(Q)^\dagger$,
- $Q =_\beta Q'$ implies $(Q)^\dagger =_\beta (Q')^\dagger$.

Lemma 27 *For every closed term $Q \in \widehat{\Lambda}^{\mathrm{aff}}(\Sigma_1)$, $\tau_1'(Q) \in \mathcal{T}(\mathscr{A}_1)$ iff $(Q)^\dagger =_\beta Q =_\beta \widetilde{Q}$.*

Lemma 28 *For every closed term $P \in \Lambda^{\mathrm{aff}}(\Sigma_1)$, $|P|_\beta$ is linear iff there is $Q \in \Pi(P)$ whose type is in $\mathcal{T}(\mathscr{A}_1)$.*

Second-Order Case

We say that an abstract atomic type $p \in \mathscr{A}_0$ is *useless* if there is no $M \in \mathcal{A}(\mathscr{G})$ that has a subterm whose type contains p. An abstract constant $\mathsf{a} \in \mathscr{C}_0$ is *useless* if there is no $M \in \mathcal{A}(\mathscr{G})$ containing a. If an ACG is second-order, it is easy to check whether the abstract vocabulary contains useless atomic types or constants, and if so, we can eliminate useless abstract atomic types and constants. This can be done in a way similar to the elimination of useless nonterminal symbols and productions from a context-free grammar.

Definition 15 Let $\mathscr{G} = \langle \Sigma_0, \Sigma_1, \mathscr{L}, s \rangle$ be a second-order ACG that has no useless abstract atomic types or constants. We define $\mathscr{G}' = \langle \Sigma_0', \Sigma_1, \mathscr{L}', [s, \mathscr{L}(s)] \rangle$

as follows: define $\Sigma_0' = \langle \mathscr{A}_0', \mathscr{C}_0', \tau_0' \rangle$ by

$$\mathscr{A}_0' = \{ [p, \beta] \mid p \in \mathscr{A}_0,\ \beta \in \Pi(\mathscr{L}(p)) - \mathcal{T}(\overline{\mathscr{A}_1}) \},$$
$$\mathscr{C}_0' = \{ [\![\mathsf{a}, Q]\!] \mid \mathsf{a} \in \mathscr{C}_0,\ Q \in \Pi(\mathscr{L}(\mathsf{a})) - \Lambda^{\mathrm{aff}}(\overline{\Sigma_1}) \},$$
$$\tau_0' = \{ [\![\mathsf{a}, Q]\!] \mapsto ([\tau_0(\mathsf{a}), \tau_1'(Q)])^{\ddagger} \},$$
$$\text{where } ([p, \beta])^{\ddagger} = [p, \beta],$$
$$([\alpha \to \gamma, \beta \to \delta])^{\ddagger} = \begin{cases} ([\alpha, \beta])^{\ddagger} \to ([\gamma, \delta])^{\ddagger} & \text{if } \beta \notin \mathcal{T}(\overline{\mathscr{A}_1}), \\ ([\gamma, \delta])^{\ddagger} & \text{if } \beta \in \mathcal{T}(\overline{\mathscr{A}_1}), \end{cases}$$

and $\mathscr{L}'$ by

$$\mathscr{L}'([p, \beta]) = (\beta)^{\dagger}, \qquad \mathscr{L}'([\![\mathsf{a}, Q]\!]) = (Q)^{\dagger}.$$

$\mathscr{G}'$ is linear, but it may contain useless abstract atomic types or constants. The *linearized ACG* $\mathscr{G}^l$ for $\mathscr{G}$ is the result of eliminating all the useless abstract atomic types and constants from $\mathscr{G}'$.

Lemma 29 *Let $\mathscr{G}$ and $\mathscr{G}'$ be as in Definition 15.*

For every variable-free $M \in \Lambda^{\mathrm{lin}}(\Sigma_0)$ of an atomic type and every $Q \in \Pi(\mathscr{L}(M)) - \Lambda^{\mathrm{aff}}(\overline{\Sigma_1})$, there is $N \in \Lambda^{\mathrm{lin}}(\Sigma_0')$ such that $\tau_0'(N) = [\tau_0(M), \tau_1'(Q)]$ and $\mathscr{L}'(N) \equiv (Q)^{\dagger}$.

Conversely, for every variable-free $N \in \Lambda^{\mathrm{lin}}(\Sigma_0')$ of an atomic type, there are $M \in \Lambda^{\mathrm{lin}}(\Sigma_0)$ and $Q \in \Pi(\mathscr{L}(M)) - \Lambda^{\mathrm{aff}}(\overline{\Sigma_1})$ such that $\tau_0'(N) = [\tau_0(M), \tau_1'(Q)]$ and $\mathscr{L}'(N) \equiv (Q)^{\dagger}$.

Theorem 30 *For every affine ACG $\mathscr{G} \in \mathbf{G}^{\mathrm{aff}}(2, n)$, there is a linear ACG $\mathscr{G}^l \in \mathbf{G}^{\mathrm{lin}}(2, n)$ such that $\mathcal{O}(\mathscr{G}^l) = \{ P \in \mathcal{O}(\mathscr{G}) \mid P \text{ is linear} \}$.*

Proof. Use Lemmas 28, 29, and 27. □

De Groote and Pogodalla (2003, 2004) have presented encoding methods for linear CFTGs and LCFRSs by linear ACGs. Their methods can also be applied to non-duplicating CFTGs and MCFGs.

Example 2 Let a non-duplicating CFTG G consist of the following productions:[4]

$$S \to P(\mathsf{a}, \mathsf{b}), \quad P(x_1, x_2) \to P(\mathsf{c}(x_1), \mathsf{c}(S)) \mid \mathsf{d}(x_1, x_2),$$

where the ranks of S, P, a, b, c, d are 0, 2, 0, 0, 1, 2, respectively. De Groote and Pogodalla's method transforms G into the following affine ACG $\mathscr{G}$:

$\mathsf{x} \in \mathscr{C}_0$	$\tau_0(\mathsf{x})$	$\mathscr{L}(\mathsf{x})$
A	$p \to s$	$\lambda y_p^{o^2 \to o}.y_p \mathsf{a}^o \mathsf{b}^o$
B	$s \to p \to p$	$\lambda y_s^o y_p^{o^2 \to o} x_1^o x_2^o.y_p(\mathsf{c}^{o \to o} x_1)(\mathsf{c}^{o \to o} y_s)$
C	p	$\lambda x_1^o x_2^o.\mathsf{d}^{o^2 \to o} x_1 x_2$

[4]The notation adopted here follows de Groote and Pogodalla.

When we apply the linearization method given in Definition 15 to $\mathscr{G}$, we get the following linear ACG $\mathscr{G}^l$ whose distinguished type is $[s, o]$:

$\mathsf{x} \in \mathscr{C}_0^l$ $\tau_0^l(\mathsf{x})$	$\mathscr{L}^l(\mathsf{x})$
$[\![\mathsf{A}, \lambda y_p^{o\to o\to o}.y_p\mathsf{ab}]\!]$ $[p, o \to o \to o] \to [s, o]$	$\lambda y_p^{o^2\to o}.y_p\mathsf{ab}$
$[\![\mathsf{A}, \lambda y_p^{o\to\bar{o}\to o}.y_p\mathsf{a}\bar{\mathsf{b}}]\!]$ $[p, o \to \bar{o} \to o] \to [s, o]$	$\lambda y_p^{o\to o}.y_p\mathsf{a}$
$[\![\mathsf{B}, \lambda y_s^o y_p^{o\to o\to o} x_1^o x_2^{\bar{o}}.y_p(\mathsf{c}x_1)(\mathsf{c}y_s)]\!]$ $[s, o] \to [p, o \to o \to o] \to [p, o \to \bar{o} \to o]$	$\lambda y_s^o y_p^{o^2\to o} x_1^o.y_p(\mathsf{c}x_1)(\mathsf{c}y_s)$
$[\![\mathsf{B}, \lambda y_s^{\bar{o}} y_p^{o\to\bar{o}\to o} x_1^o x_2^{\bar{o}}.y_p(\mathsf{c}x_1)(\bar{\mathsf{c}}y_s)]\!]$ $[p, o \to \bar{o} \to o] \to [p, o \to \bar{o} \to o]$	$\lambda y_p^{o\to o} x_1^o.y_p(\mathsf{c}x_1)$
$[\![\mathsf{C}, \lambda x_1^o x_2^o.\mathsf{d}x_1x_2]\!]$ $[p, o \to o \to o]$	$\lambda x_1^o x_2^o.\mathsf{d}x_1x_2$

The linearized ACG $\mathscr{G}^l$ is actually the encoding of the linear CFTG G' consisting of the following productions:

$$S \to P(\mathsf{a}, \mathsf{b}) \mid P'(\mathsf{a}), \quad P'(x_1) \to P(\mathsf{c}(x_1), \mathsf{c}(S)) \mid P'(\mathsf{c}(x_1)),$$
$$P(x_1, x_2) \to \mathsf{d}(x_1, x_2),$$

where the ranks of nonterminals S, P, P' are 0, 2, 1, respectively. G, $\mathscr{G}$, $\mathscr{G}^l$, and G' generate the same tree language.

The following corollary generalizes the result by Fujiyoshi (2005), which covers the *monadic* case only.

Corollary 31 *For every non-duplicating CFTG G, there is a linear CFTG G' such that G and G' generate the same tree language.*

Let $\mathscr{G}$ be the affine ACG that encodes an MCFG G. The linearized ACG $\mathscr{G}^l$ is indeed in the form that is the encoding of an LCFRS[5] (but $\mathscr{G}'$ is not). Therefore, our result covers the following theorem shown by Seki et al. (1991).

Corollary 32 *For every MCFG G, there is an LCFRS G' such that the languages generated by G and G' coincide.*

Third or Higher-Order Case

Definition 15 itself does not depend on the order of the given affine ACG except that in the general case, we do not know how to find and eliminate useless abstract atomic types and constants. For the general case, however, the linearized ACG given in Definition 15 may generate a strictly larger language

[5] The LCFRS obtained from an MCFG through our linearization method may have nonterminals of rank 0. The reason why usual definitions of an LCFRS do not allow nonterminals to have rank 0 is just to avoid redundancy. Mathematically speaking, allowing or disallowing nonterminals of rank 0 does not matter at all.

than the original affine ACG. In the remainder of this paper, we present a linearization method for general affine ACGs.

Example 3 Suppose that an affine ACG $\mathscr{G} \in \mathbf{G}^{\mathrm{aff}}(3, 1)$ consists of the following lexical entries:

$\mathsf{x} \in \mathscr{C}_0$	$\tau_0(\mathsf{x})$	$\mathscr{L}(\mathsf{x})$
A	q	$\#$
B	$p \to q \to q$	$\lambda y^o z^o.\mathsf{b}^{o\to o} z$
C	$q \to s$	$\lambda z^o.z$
D	$(p \to s) \to s$	$\lambda x^{o\to o}.\mathsf{a}^{o\to o}(x\mathsf{e}^o)$

We see $O(\mathscr{G}) = \{\, \overbrace{\mathsf{a}(\ldots(\mathsf{a}}^{n\text{-times}}(\overbrace{\mathsf{b}(\ldots(\mathsf{b}}^{n\text{-times}}\,\#)\ldots))\ldots) \mid n \geq 0 \,\}$. The linear ACG $\mathscr{G}'$ by Definition 15 consists of the following lexical entries:

$\mathsf{x} \in \mathscr{C}'_0$	$\tau'_0(\mathsf{x})$	$\mathscr{L}'(\mathsf{x})$
$[\![\mathsf{A}, \#]\!]$	$[q, o]$	$\#$
$[\![\mathsf{B}, \lambda y^{\overline{o}} z^o.\mathsf{b}z]\!]$	$[q, o] \to [q, o]$	$\lambda z^o.\mathsf{b}z$
$[\![\mathsf{C}, \lambda z^o.z]\!]$	$[q, o] \to [s, o]$	$\lambda z^o.z$
$[\![\mathsf{D}, \lambda x^{\overline{o}\to o}.\mathsf{a}(x\overline{\mathsf{e}})]\!]$	$[s, o] \to [s, o]$	$\lambda x^o.\mathsf{a}x$
$[\![\mathsf{D}, \lambda x^{o\to o}.\mathsf{a}(x\mathsf{e})]\!]$	$([p, o] \to [s, o]) \to [s, o]$	$\lambda x^{o\to o}.\mathsf{a}(x\mathsf{e})$

The last lexical entry is useless. We have

$$O(\mathscr{G}') = \{\, \overbrace{\mathsf{a}(\ldots(\mathsf{a}}^{m\text{-times}}(\overbrace{\mathsf{b}(\ldots(\mathsf{b}}^{n\text{-times}}\,\#)\ldots))\ldots) \mid m, n \geq 0 \,\} \supsetneq O(\mathscr{G}).$$

Though any term of type p that is the first argument of an occurrence of B is bound to be erased in the original ACG $\mathscr{G}$, we cannot ignore the occurrence of the type p, because that occurrence of p balances the numbers of occurrences of B and D in a term in $\mathcal{A}(\mathscr{G})$.

Our new linearization method gives the linear ACG $\mathscr{G}''$ consisting of the following lexical entries (useless lexical entries are suppressed):

$\mathsf{x} \in \mathscr{C}''_0$	$\tau''_0(\mathsf{x})$	$\mathscr{L}''(\mathsf{x})$
$[\![\mathsf{A}, \#]\!]$	$[q, o]$	$\#$
$[\![\mathsf{B}, \lambda y^{\overline{o}} z^o.\mathsf{b}z]\!]$	$[p, \overline{o}] \to [q, o] \to [q, o]$	$\lambda y^{o\to o} z^o.y(\mathsf{b}z)$
$[\![\mathsf{C}, \lambda z^o.z]\!]$	$[q, o] \to [s, o]$	$\lambda z^o.z$
$[\![\mathsf{D}, \lambda x^{\overline{o}\to o}.\mathsf{a}(x\overline{\mathsf{e}})]\!]$	$([p, \overline{o}] \to [s, o]) \to [s, o]$	$\lambda x^{(o\to o)\to o}.\mathsf{a}(x(\lambda z^o.z))$

where $[p, \overline{o}]$ is mapped to $o \to o$. We have $O(\mathscr{G}) = O(\mathscr{G}'')$.

Now, we give the formal definition of our new linearization method for general affine ACGs. For simplicity, we assume that $\mathscr{A}_1 = \{o\}$ here, but it is possible to lift this assumption. The new linearized ACG $\mathscr{G}''$ has the form

$\mathscr{G}'' = \langle \Sigma_0'', \Sigma_1, \mathscr{L}'', [s, \mathscr{L}(s)]\rangle$, where $\Sigma_0'' = \langle \mathscr{A}_0'', \mathscr{C}_0'', \tau_0''\rangle$ is defined by

$$\begin{aligned} \mathscr{A}_0'' &= \{\, [p,\beta] \mid p \in \mathscr{A}_0,\ \beta \in \Pi(\mathscr{L}(p)) \,\}, \\ \mathscr{C}_0'' &= \{\, [\![\mathsf{a}, Q]\!] \mid \mathsf{a} \in \mathscr{C}_0,\ Q \in \Pi(\mathscr{L}(\mathsf{a})) \,\}, \\ \tau_0'' &= \{\, [\![\mathsf{a}, Q]\!] \mapsto [\tau_0(\mathsf{a}), \tau_1'(Q)] \,\} \\ &\quad \text{where } [\alpha \to \gamma, \beta \to \delta] = [\alpha,\beta] \to [\gamma,\delta]. \end{aligned}$$

Here we have two simple lexicons $\mathscr{L}_0 : \Sigma_0'' \to \Sigma_0$ and $\mathscr{L}_1 : \Sigma_0'' \to \Sigma_1'$;

$$\mathscr{L}_0([p,\beta]) = p, \quad \mathscr{L}_0([\![\mathsf{a}, Q]\!]) = \mathsf{a}, \quad \mathscr{L}_1([p,\beta]) = \beta, \quad \mathscr{L}_1([\![\mathsf{a}, Q]\!]) = Q.$$

We have $\widetilde{\mathscr{L}_1(N)} \equiv \mathscr{L} \circ \mathscr{L}_0(N)$ for $N \in \Lambda^{\mathrm{lin}}(\Sigma_0'')$. For any $M \in \Lambda^{\mathrm{lin}}(\Sigma_0)$ and $Q \in \Pi(\mathscr{L}(M))$, one can find a term $\chi(M,Q) \in \Lambda^{\mathrm{lin}}(\Sigma_0'')$ such that $\mathscr{L}_0(\chi(M,Q)) \equiv M$ and $\mathscr{L}_1(\chi(M,Q)) \equiv Q$.

Lemma 33 *For every $Q \in \widehat{\Lambda}^{\mathrm{aff}}(\Sigma_1)$ and $\alpha \in \mathcal{T}(\mathscr{A}_0)$, the following statements are equivalent:*

1. *There is $M \in \Lambda^{\mathrm{lin}}(\Sigma_0)$ of type α such that $\mathscr{L}(M) \equiv \widetilde{Q}$.*
2. *There is $N \in \Lambda^{\mathrm{lin}}(\Sigma_0'')$ of type $[\alpha, \tau_1'(Q)]$ such that $\mathscr{L}_1(N) \equiv Q$.*

Lemmas 28 and 33 imply

$$\{\, M \in \mathcal{A}(\mathscr{G}) \mid |\mathscr{L}(M)|_\beta \text{ is linear} \,\} = \{\, \mathscr{L}_0(N) \mid N \in \mathcal{A}(\mathscr{G}'') \,\}.$$

Since $(\mathscr{L}_1(N))^\dagger =_\beta \widetilde{\mathscr{L}_1(N)} \equiv \mathscr{L} \circ \mathscr{L}_0(N)$ for every $N \in \mathcal{A}(\mathscr{G}'')$ by Lemma 27, it is enough to define a new lexicon $\mathscr{L}''$ so that

$$\mathscr{L}''(N) =_{\beta\eta} (\mathscr{L}_1(N))^\dagger \tag{12.12}$$

for every $N \in \mathcal{A}(\mathscr{G}'')$.

We define the type substitution $\sigma : \mathscr{A}_0'' \to \mathcal{T}(\{o\})$ of $\mathscr{L}'' = \langle \sigma, \theta\rangle$ as

$$\sigma([p,\beta]) = \begin{cases} (\beta)^\dagger & \text{if } \beta \notin \mathcal{T}(\{\overline{o}\}), \\ o \to o & \text{if } \beta \in \mathcal{T}(\{\overline{o}\}). \end{cases}$$

Here we identify σ with its homomorphic extension. As a preparation for defining the term substitution θ of $\mathscr{L}''$, we give three kinds of linear combinators. For $[\alpha,\beta] \in \mathcal{T}(\mathscr{A}_0'')$ such that $\beta \in \mathcal{T}(\{\overline{o}\})$, let $\sigma([\alpha,\beta]) = \gamma_1 \to \cdots \to \gamma_m \to o \to o$ and $\gamma_i = \gamma_{i,1} \to \cdots \to \gamma_{i,k_i} \to o \to o$. $Z^{\sigma([\alpha,\beta])}$ is a linear combinator of type $\sigma([\alpha,\beta])$ defined as

$$\begin{aligned} Z^{\sigma([\alpha,\beta])} &\equiv \lambda y_1^{\gamma_1} \ldots y_m^{\gamma_m} z^o.R_1(R_2(\ldots(R_m z)\ldots)) \\ &\quad \text{where } R_i \equiv y_i^{\gamma_i} Z^{\gamma_{i,1}} \ldots Z^{\gamma_{i,k_i}}. \end{aligned}$$

For each $[\alpha,\beta] \in \mathcal{T}(\mathscr{A}_0'')$ such that $\beta \in \widehat{\mathcal{T}}(\{o\}) - \mathcal{T}(\{\overline{o}\})$, we define two linear combinators X_α^β of type $\sigma([\alpha,\beta]) \to (\beta)^\dagger$ and Y_α^β of type $(\beta)^\dagger \to \sigma([\alpha,\beta])$ by mutual induction. Let $[\alpha,\beta] = [\alpha_1,\beta_1] \to \cdots \to [\alpha_m,\beta_m] \to [p,\beta_0]$ with $[p,\beta_0] \in \mathscr{A}_0''$ and the set $\{1,\ldots,m\}$ be partitioned into two subsets I and J so that $\beta_i \notin \mathcal{T}(\{\overline{o}\})$ iff $i \in I$. Let $I = \{i_1, \ldots, i_k\}$ ($i_j < i_{j+1}$) and $J = \{j_1, \ldots, j_l\}$. Let

$$X_\alpha^\beta \equiv \lambda y^{\sigma([\alpha,\beta])} x_{i_1}^{(\beta_{i_1})^\dagger} \ldots x_{i_k}^{(\beta_{i_k})^\dagger}.y^{\sigma([\alpha,\beta])} P_1 \ldots P_m$$

$$\text{where} \quad P_i \equiv \begin{cases} Y_{\alpha_i}^{\beta_i} x_i^{(\beta_i)^\dagger} & \text{if } i \in I, \\ Z^{\sigma([\alpha_i,\beta_i])} & \text{if } i \in J, \end{cases}$$

and

$$Y_\alpha^\beta \equiv \lambda x^{(\beta)^\dagger} y_1^{\sigma([\alpha_1,\beta_1])} \ldots y_m^{\sigma([\alpha_m,\beta_m])} \vec{z}.$$

$$M_{j_1}(\ldots(M_{j_l}(x^{(\beta)^\dagger} L_{i_1} \ldots L_{i_k} \vec{z}))\ldots)$$

where $\vec{z}$ is short for $z_1^{\gamma_1} \ldots z_n^{\gamma_n}$ for $(\beta_0)^\dagger = \gamma_1 \to \cdots \to \gamma_n \to o$, and

$$\begin{cases} L_i \equiv X_{\alpha_i}^{\beta_i} y_i^{\sigma([\alpha_i,\beta_i])} & \text{for } i \in I, \\ M_i \equiv Z^{\sigma([\alpha_i,\beta_i]) \to o \to o} y_i^{\sigma([\alpha_i,\beta_i])} & \text{for } i \in J. \end{cases}$$

Note that if $[\alpha,\beta] = [p,\beta_0] \in \mathscr{A}_0''$, then $X_p^{\beta_0} =_{\beta\eta} Y_p^{\beta_0} =_{\beta\eta} \lambda z^{(\beta_0)^\dagger}.z$.

Now, we give a new linearization method as follows.

Definition 16 For a given affine ACG $\mathscr{G}$, we define a new linear ACG as $\mathscr{G}'' = \langle \Sigma_0'', \Sigma_1, \mathscr{L}'', [s, \mathscr{L}(s)] \rangle$, where $\mathscr{L}'' = \langle \sigma, \theta \rangle$ for σ as above and

$$\theta([\![\mathsf{a}, Q]\!]) \equiv \begin{cases} |Y_{\tau_0(\mathsf{a})}^{\tau_1'(Q)}(Q)^\dagger|_\beta & \text{if } \tau_1'(Q) \notin \mathcal{T}(\{\overline{o}\}), \\ Z^{\sigma(\tau_0''([\![\mathsf{a},Q]\!]))} & \text{if } \tau_1'(Q) \in \mathcal{T}(\{\overline{o}\}). \end{cases}$$

If $\mathscr{G} \in \mathbf{G}^{\mathrm{aff}}(m,n)$, then $\mathscr{G}'' \in \mathbf{G}^{\mathrm{lin}}(m, \max\{2,n\})$.

Lemma 34 *Given $N \in \Lambda(\Sigma_0'')$ of type $[\alpha,\beta]$ such that $\beta \notin \mathcal{T}(\{\overline{o}\})$ and $\mathscr{L}_1(N) \in \widehat{\Lambda}^{\mathrm{aff}}(\Sigma_1)$, we have*

$$(\mathscr{L}_1(N))^\dagger =_{\beta\eta} X_\alpha^\beta \mathscr{L}''(N)\phi_N$$

where ϕ_N is the substitution on the free variables of $\mathscr{L}''(N)$ such that

$$x^{\sigma([\alpha,\beta])}\phi_N = \begin{cases} Y_\alpha^\beta x^{(\beta)^\dagger} & \text{if } x \text{ has the type } [\alpha,\beta] \text{ in } N \text{ and } \beta \notin \mathcal{T}(\{\overline{o}\}), \\ Z^{\sigma([\alpha,\beta])} & \text{otherwise.} \end{cases}$$

Theorem 35 *For every affine ACG $\mathscr{G} \in \mathbf{G}^{\mathrm{aff}}(m,n)$, there is a linear ACG $\mathscr{G}'' \in \mathbf{G}^{\mathrm{lin}}(m, \max\{2,n\})$ such that $\mathcal{O}(\mathscr{G}'') = \{ P \in \mathcal{O}(\mathscr{G}) \mid P \text{ is linear} \}$.*

Proof. Lemma 34 entails the equation (12.12). □

12.4 Concluding Remarks

We have shown that the generative capacity of linear ACGs is as rich as that of affine ACGs, that is, the non-deletion constraint on linear ACGs is superficial. Our linearization method, however, increases the size of the given grammar exponentially due to the definition of Π, so there may still exist an advantage of allowing deleting operations in the ACG formalism. For instance, the atomic type *np* of the abstract vocabulary of the ACG in Example 1 will be divided up into three new atomic types which correspond to noun phrases as third person singular subjects, plural subjects, and objects, respectively.

One attractive feature of ACGs is that they can be thought of as a generalization of several well-established grammar formalisms (de Groote, 2002, de Groote and Pogodalla, 2003, 2004). This paper demonstrates that the ACG formalism also generalizes some "operation" on those grammars, namely, conversion from non-duplicating grammars into non-duplicating and non-deleting ones. Recall that Fisher (1968a,b) showed that every CFTG has a corresponding non-deleting CFTG whose string IO-language is equivalent. One may wonder if Fisher's result can be generalized to a transformation from λK-*ACGs*, where duplicating operations are allowed as well as deleting ones, into equivalent λI-*ACGs*, where duplicating operations are allowed but deleting ones are not. That is future work. The author conjectures that one can eliminate vacuous λ-abstraction from *semi-affine ACGs*, where a term is *semi-affine* if for every free variable x of any subterm, either x occurs at most once, or x has an atomic type. Actually, every CFTG has a corresponding semi-affine ACG such that the tree IO-language of the CFTG coincides with the object language of the ACG, and the semi-affine ACG encoding a non-deleting CFTG has no vacuous λ-abstraction. If the conjecture is correct, this implies that every CFTG has a corresponding non-deleting CFTG whose tree IO-language is equivalent. This also entails Fisher's result.

References

de Groote, Philippe. 2001. Towards abstract categorial grammars. In *Association for Computational Linguistics, 39th Annual Meeting and 10th Conference of the European Chapter, Proceedings of the Conference*, pages 148–155.

de Groote, Philippe. 2002. Tree-adjoining grammars as abstract categorial grammars. In *TAG+6, Proceedings of the 6th International Workshop on Tree Adjoining Grammars and Related Frameworks*, pages 145–150. Università di Venezia.

de Groote, Philippe and Sylvain Pogodalla. 2003. *m*-linear context-free rewriting systems as abstract categorial grammars. In R. T. Oehrle and J. Rogers, eds., *Proceedings of Mathematics of Language - MOL-8, Bloomington, Indiana, U. S.*, pages 71–80.

de Groote, Philippe and Sylvain Pogodalla. 2004. On the expressive power of abstract categorial grammars: Representing context-free formalisms. *Journal of Logic, Language and Information* 13(4):421–438.

Fisher, Michael J. 1968a. *Grammars with Macro-Like Productions*. Ph.D. thesis, Harvard University.

Fisher, Michael J. 1968b. Grammars with macro-like productions. In *Proceedings of the 9th IEEE Conference on Switching and Automata Theory*, pages 131–142.

Fujiyoshi, Akio. 2005. Linearity and nondeletion on monadic context-free tree grammars. *Information Processing Letters* 93(3):103–107.

Hindley, J. Roger. 1997. *Basic Simple Type Theory*. Cambridge University Press.

Kanazawa, Makoto and Ryo Yoshinaka. 2005. Lexicalization of second-order ACGs. Tech. Rep. NII-2005-012E, National Institute of Informatics.

Salvati, Sylvain. 2006. Encoding second order string ACGs with deterministic tree walking transducers. In *Proceedings of the 11th conference on Formal Grammar*.

Seki, Hiroyuki, Takashi Matsumura, Mamoru Fujii, and Tadao Kasami. 1991. On multiple context-free grammars. *Theoretical Computer Science* 88(2):191–229.