TYPE-DRIVEN NATURAL LANGUAGE ANALYSIS

Remo Pareschi

PhD

University of Edinburgh

1988

I hereby declare that this thesis was written by me, and that the research reported in there is my own.

Abstract

The purpose of this thesis is in showing how recent developments in logic programming can be exploited to encode in a computational environment the features of certain linguistic theories. We are in this way able to make available for the purpose of natural language processing sophisticated capabilities of linguistic analysis directly justified by well developed grammatical frameworks.

More specifically, we exploit hypothetical reasoning, recently proposed as one of the possible directions to widen logic programming, to account for the syntax of filler-gap dependencies along the lines of linguistic theories such as Generalized Phrase Structure Grammar and Categorial Grammar. Moreover, we make use, for the purpose of semantic analysis of the same kind of phenomena, of another recently proposed extension, interestingly related to the previous one, namely the idea of replacing first-order terms with the more expressive λ-terms of λ-Calculus.

Contents

List of Figures

Preface

This thesis brings together my interest in natural language processing and in the use of logic as a tool for computation. As such, its development has been influenced by several people.

Mark Steedman acted both as a supervisor and as a friend, providing insightful advice and moral support. His constant encouragement has been essential to the accomplishment of the enterprise. To him, and to Lauri Karttunen before him, I owe my interest in the relationship between grammatical formalisms and computation.

Dale Miller unveiled for me the subtleties of proof theory, and answered patiently all my questions on the subject. His ideas on logic programming have been the foremost influence on this research.

I am indebted to Ewan Klein, who was my examiner together with Dale, for a number of important improvements which have been incorporated in the final draft.

Henry Thompson's concern with the clear communication of concepts and ideas had a most relevant impact upon me, which is one of the reasons why this thesis is a bit less un-understandable than what it would have been had I written it not today, but four years ago.

I am grateful to Aravind Joshi, Fernando Pereira and David Weir for illuminating discussions. Lynette Hirschmann and Esther König invited me to give talks at the Unisys Research Center in Paoli, Pennsylvania, and at the IBM Research Center in Stuttgart, which allowed me to present some of the results

of my research to sympathetic and stimulating audiences. I now realize how important conversations with Jo Calder, Nick Haddock, Mark Hepple, Einar Jowsey, Glyn Morrill, Michael Moortgat, Stu Shieber, Mary Woods and Henk Zeevat have been to me, even if many of them took place at a time when the ideas behind this dissertation were still less than "half-baked". Thanks are also due to Erhard Hinrichs, Mitch Marcus, David McCarty, Marc Moens, Barry Richards, Bonnie Webber and Carl Weir—even though they may not realize why.

I owe a special debt to Elsa Gunter. During the year in which we shared the same office at University of Pennsylvania, I could take advantage of her expertise in the use of Lambda Prolog, of her competent knowledge of Latex and Tex and, more than anything else, of her generous assistance in the typesetting of the first draft of this thesis, which led her to spend an extraordinary amount of time in helping me to meet several overlapping deadlines. I also wish to thank Andy Dwelly for help in the typesetting of the final draft.

I am grateful to Aravind Joshi and Mark Steedman, who made possible for me to visit the Department of Computer and Information Sciences at University of Pennsylvania during the academic year 1987-88. This visit has been quite essential for the development of this research. Finally, I wish to thank my current employer, the European Computer-Industry Research Center, in the persons of its director, Hervé Gallaire, and the director of the Logic Programming group, Alexander Herold, for allowing me to work on the revisions which produced the final draft of this thesis.

I also gratefully acknowledge the partial financial support of an Edinburgh University Postgraduate Studentship for the years 1984-87, and, for the year 1987-88, of a Sloan Foundation grant to the Cognitive Science Program, University of Pennsylvania, and NSF grants MCS-8219196, IRI-10413 AO2, ARO grants DAA29-84-K-0061, DAA29-84-90027 and DARPA grant NOOO14-85-K0018 to the Department of Computer and Information Sciences at University of Pennsylvania.

Chapter 1

Introduction

What might have been is an abstraction
Remaining a perpetual possibility
Only in a world of speculation.
What might have been and what has been
Point to one end, which is always present.
Footfalls echo in the memory
Down the passage we did not take
Towards the door we never opened
Into the rose-garden. My words echo
Thus, in your mind.

– T. S. Eliot, Burnt Norton, (1935)

1.1 Motivations

1.1.1 Syntactic Analysis of Extraction Phenomena

This thesis is made possible by certain deep and subtle results in proof-theory, but its motivations are informally justifiable in terms of an intuitively easy and simple leitmotiv:

the accounts of extraction phenomena in natural language proposed by certain linguistic theories can be logically and computationally reconstructed by extending the definite clause logic behind the logic programming language Prolog to allow implications as goals and as internal parts of definite clauses.

Thus, consider for instance the following phrase structure rule from the framework of Generalized Phrase Structure Grammar:

$$REL \rightarrow \textbf{whom} \quad S/NP$$

The informal characterization of this phrase structure rule is that a relative clause is given by the relative pronoun **whom** followed by something which amounts to a sentence missing somewhere a noun phrase. Such a characterization can even more perspicuously be stated via the following form of hypothetical reasoning: if the relative pronoun *whom* is followed by something which *if there were a noun phrase* would be a sentence then we have a relative clause. Formally, this can be accounted for by translating the rule above into a definite clause, according to the familiar methodology developed in the framework of Definite Clause Grammars, with the addition that we make the slashed category inside the rule an internal implication of the definite clause. Thus, abstracting from the arguments which will be passed to the predicates corresponding to the non-terminals, and choosing for the time being a simple propositional representation, the rule above can be translated in the following formula:

$$(\textbf{whom} \ \wedge \ (NP \ \supset \ S)) \ \supset \ REL$$

Or consider the following assignment of a syntactic type from the framework of Categorial Grammar:

$$\textbf{whom} \ : \ REL/(S/NP)$$

What this type assignment says is that the relative pronoun **whom** can be viewed as a function taking functions from noun phrases to sentences and returning relative clauses. Thus, we can have an analogous encoding of such a type as a definite clause with an internal implication corresponding to the type

of the input function. Again, staying for the time being at the level of propositional logic, this implies translating the type assignment above into something of the following kind:

$$\textbf{whom} \ : \ (NP \supset S) \supset REL$$

Of course, we are not the first ones to discover the analogy between the slash notation / used by linguists and the implication connective used in logic. Indeed, such an analogy has a long and venerable history, perhaps culminated in Joachim Lambek's effort in the late fifties to produce a logical calculus where the meaning of the slash as used by linguists in Categorial Grammar is in fact captured proof-theoretically as a form of implication. Our effort here is directly related to Lambek's enterprise, although the attitude we take is very different: rather than viewing the slash as a specialized form of implication, we show that by replacing the slash with standard implication one obtains a formalism of increased expressive power, which can be elegantly incorporated in recently proposed extensions of logic programming.

1.1.2 Semantic Analysis of Extraction Phenomena

But indeed we have also a second leitmotiv, perhaps more obvious but certainly as important as the one we started with:

the natural semantic interpretation of an expression missing an internal constituent is in the form of a λ-abstraction

We say that this is an obvious statement because there is an almost unanimous agreement about it across several linguistic theories; however, a framework like Definite Clause Grammars so far has been unable to account for it in a formally adequate way, since it presupposes a logic built around first-order terms, rather than λ-terms.

There is a recent development of logic programming in the direction of constructive logic which wants to replace Prolog with a language at the same time more pure and more powerful. Such a development contemplates the use both of internal implications and of λ-terms, and has been carried out from the point of

view of practical implementation in the logic programming language λ-Prolog. We shall show in the course of this thesis that this new paradigm for logic programming offers the ideal vehicle to implement the linguistic features mentioned above, thus rekindling in an interesting way the dialogue between logic as a tool for computation and natural language processing.

1.2 Outline of Contents

In Chapter 2 we give a formal and conceptual characterization of the constructive extension of Prolog which we shall exploit for the purpose of linguistic analysis in the rest of the thesis. In Chapter 3 we show how this extension allows us to enrich the Definite Clause Grammars framework with features coming from Generalized Phrase Structure Grammar. In Chapter 4 we examine the technique of gap threading, developed to deal with cases of extraction by the tradition of logic grammars, and we redefine it in a logically cleaner way in our formalism. In Chapter 5 we give a definite clause version of Categorial Grammar. Finally, in chapter 6 we deal with certain issues concerning the implementation of the ideas developed in the previous chapters.

Chapter 2

A Constructive Paradigm for Logic Programming

2.1 Introduction

Logic programming and natural language processing have in the past been fruitfully indebted to each other. Prolog, so far the most successful example of a logic programming language, came out in the early seventies as a result of Alain Colmerauer's efforts to create a programming environment suitable both for natural language processing and deductive question-answering. In the early eighties Pereira and Warren [43] gave a rigorous definition of the framework of Definite Clause Grammars (DCGs), which is directly inspired by the possibility of encoding phrase-structure grammars as Prolog programs. DCGs represent a fundamental contribution to the formalization of linguistic theories for computational purposes, and to the idea that grammatical formalisms can be viewed as programming languages.

Nowadays, a significant portion of the research in logic programming is in the direction of extending Prolog towards a notion of logic programming language at the same time more pure and more powerful, without losing the procedural efficiency which is one of the key reasons of Prolog's success. Particularly important from this point of view are the efforts of Dale Miller and his associates [29, 31, 32, 33, 34, 30], who extend Prolog in the direction of Intuitionistic Logic

while maintaining Prolog's crucial feature of returning *definite* answer substitutions as output of a logic program. The extended notion of logic programming language they introduce allows for *implicational* goals as well as for *generic* (i. e., universally quantified) goals; such extensions can be used for elegant implementations of modules and abstract datatypes [29, 30]. They also go beyond the use of first-order terms, and they replace them with λ-terms, thus introducing certain features from higher-order logic and λ-Calculus, like unification with built-in β-reduction, which allow elegant data-manipulations of formula and programs. While still remaining in the realm of first-order logic, McCarty [27, 28] also develops a similar framework, and uses it for hypothetical reasoning in knowledge representation, and Gabbay [8] defines an extended Prolog of this kind also for the purpose of hypothetical reasoning, and of metareasoning.

One of the goals of this thesis is to show that this extended notion of logic programming offers the opportunity of a new encounter between logic programming and natural language processing, such that some of the latest developments of modern linguistic theories can be naturally encorporated in a computational environment. In particular, the elegant accounts of extraction phenomena in natural language proposed by grammatical theories like Categorial Grammar (CG) and Generalized Phrase Structure Grammar (GPSG) can be themselves implemented in terms of hypothetical reasoning and metareasoning, via a corresponding extension of the DCG framework; this translates directly into an increased quality of the natural language analysis possible in a logic programming environment, since a crucial range of natural language phenomena are now within direct reach of the inferential machinery. We show also that existing techniques coming from the tradition of logic grammars, like the *gap threading* method for the analysis of filler-gap dependencies, can be reimplemented within this framework in a more refined and genuinely logical manner. Moreover, we make use of λ-terms to build semantic representations of the sentences being parsed, thus drawing in the spirit of Montague's enterprise, based on the idea that λ-Calculus offers a formally well-defined environment which is expressive enough for the semantic interpretation of natural language.

We introduce in this chapter an intuitionistic higher-order extension of Horn

Logic which provides the theoretical backbone for implications-as-goals and generics. In the rest of this thesis, we exploit such extensions to Horn Logic for a logical reconstruction of the treatment of extraction in CG and GPSG, in terms of an extended version of DCGs, and we reexamine under the same perspective the technique of gap threading.

2.2 Hereditary Harrop Logic

The extended notion of logic programming language we are presenting here has been baptized *Hereditary Harrop Logic* in [34]. Such a logic extends the Horn Clause Logic underlying Prolog by allowing implications-as-goals and universally quantified goals; moreover, it allows λ-abstraction over terms in the language, and quantification over function symbols, thus making use of higher-order features which are already directly supported by the logic programming language λ-Prolog [33], a variant of Prolog where unification has built-in β-reduction. The possibility of quantifying over predicate variables is another higher-order feature considered in [34]; such a feature is not of direct interest here, and will therefore be ignored, although it may have an important use for defining compact representations of coordinate structures [1]. The proof theory for Hereditary Harrop Logic is based on the *sequent systems* developed by Gentzen [12]. Sequent systems can be used for the proof theory of Classical Logic, as well as of constructive kinds of logic like Intuitionistic Logic. The usual proof theory behind Prolog theorem provers is instead that provided

[1]The name Hereditary Harrop Logic is historically justified by the fact that this logic is itself part of Harrop Logic, a larger subset of Intuitionistic Logic studied, among others, by R. Harrop [15]. Harrop Logic provides the basis for the notion of *uniform proof*, which is used in [34] to abstractly define the notion of *goal-directed programming*: uniform proofs are proofs where logical connectives directly implement the goal-directed search operations that we shall associate with the proof rules of Hereditary Harrop Logic. Although proofs in unrestricted Harrop Logic happen to be uniform "at the root", they may contain subproofs which are not uniform; hence the need of stronger conditions on the syntactic structure of formulae to ensure "hereditarity" of the uniformity property as in Hereditary Harrop Logic. See [34] for a rigourous characterization of uniform proofs.

by the technique of *resolution*, which is committed to Classical Logic, being a method of reasoning by contradiction. We shall see how sequent systems can provide an alternative, constructive account of Prolog programs, of which Hereditary Harrop Logic programs can be viewed as natural extensions.

2.2.1 Syntax

We define here the syntax of the logic programming language of Hereditary Harrop Logic, henceforth *hhl*, the constructive extension of Prolog based on Hereditary Harrop Logic, as in [29, 33, 34]. We show that this language effectively extends Prolog by first defining the language of Horn Logic, and then extending the definition itself by adding more possibilities for legally combining well-formed formulae.

2.2.2 Definite Clauses and Goal Clauses

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

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

Here, we call *ground* a clause not containing variables. We refer to the part of a non-atomic clause coming to the left of the implication connective as the *body* of the clause, and to the one on the right as the *head*.

Observe that in the definition above the notion of goal clause does not depend recursively on that of definite clause, although the set of all definite clauses is defined recursively in terms of the set of goal clauses. The logic language thus defined corresponds to that of Horn Logic, and departs from Prolog just in the

fact that it allows use of explicit quantifiers, which in principle are not needed because internal existential quantifiers in definite clauses can be mapped into external universal quantifiers, according to certain well-known logical equivalences.

Suppose now that we extend the definition above by making the notion of definite clause and the notion of goal clause mutually recursive, as follows:

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

Clearly, this extension of the language of Horn Logic allows implications-as-goals and universally quantified goals, as well as internal implications and internal universal quantifiers in definite clauses. We shall henceforth call this language *hhl*, for Hereditary Harrop Logic. In contrast to Horn Logic, in *hhl* explicit quantifiers are necessary, as we need to distinguish between universally and existentially quantified goals.

2.2.3 Lambda Terms

We follow here the following convention: constant symbols either start with capital letters, or are in boldface font; all other symbols are variables. We assume that terms in *hhl* are λ-terms as in λ-Calculus, rather than first-order terms. Under this approach, a natural representation of quantifiers is as operators over λ-expressions, as in Church's Simple Theory of Types. Therefore, a formula of the form

$$\forall x[(P\ \ x)\ \ \wedge\ \ (Q\ \ x)]$$

can be thought of as a shorthand for

$$(\forall\ \ \lambda x[(P\ \ x)\ \ \wedge\ \ (Q\ \ x)])$$

Clearly, this approach to quantification allows us to quantify over functions, so that we can have a formula such as

$$\forall f \forall x[(P\ \ (f\ \ x))\ \ \wedge\ \ (Q\ \ (f\ \ x))]$$

We shall illustrate the usefulness of λ-terms in the manipulation of the structures which can be used to encode proof trees and parse representations.

In conclusion, *hhl* increases the expressive power of standard Horn logic in several ways, the most relevant of which are the increased domain of quantification, and the fact that we permit implications and universally quantified formulae in goals and in the bodies of definite clauses.

2.2.4 Proof Theory

Logic Programs and Program Closures

We take a *logic program* or, simply, a program $\mathcal{P}$ to be any set of definite clauses. If $\mathcal{P}$ is a program, then its *closure* is the smallest set $[\mathcal{P}]$ such that:

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

Program closures provide a useful technical notion for two reasons. In first place, in virtue of *(ii)*, we can refer directly to the elements in a given conjunction of definite clauses. This will be exploited in the statement of the proof rules which define the notion of proof in our logic programming language. Moreover, in virtue of *(iii)*, we can talk about the substitution of variables with terms without going into implementational issues like unification. Thus, we shall assume that, in the concrete implementation of an interpreter for *hhl*, substitution of variables can be delayed by using unification, as in Prolog, and we shall use the word "unification" whenever this kind of situation is involved. However, in virtue of *(i)-(iii)* unification is not needed in the abstract definition of proof for *hhl*. This will make particularly simple the statement of proof rules [2].

[2]The logic oriented reader will also notice that, in the section below, the notion of program closure allows us to avoid the explicit statement of sequent inference rules for *left occurrences*

Proof Rules

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

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

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

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

(IV) $\dfrac{\mathcal{P} \Rightarrow G_i}{\mathcal{P} \Rightarrow G_1 \vee G_2}$, where $i = 1$ or $i = 2$

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

(VI) $\dfrac{\mathcal{P} \Rightarrow [x/c]G}{\mathcal{P} \Rightarrow \forall x G}$, where c is a variable which does not occur free in $\mathcal{P}$ or G

(VII) $\dfrac{\mathcal{P} \cup \{D\} \Rightarrow G}{\mathcal{P} \Rightarrow D \supset G}$

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

---

of quantifiers and connectives, that is, for occurrences of quantifiers and connectives which must be handled at the level of program clauses. In fact, we shall have explicit inference rules just for *right occurrences* of quantifiers and connectives, that is, for occurrences of quantifiers and connectives at the level of goal clauses. (The terminology *left* and *right occurrence* refers here to the position of a given occurrence with respect to the sequent arrow $\Rightarrow$.) In [29] it is shown how proofs in a sequent system of this kind can be mapped into an equivalent, more traditional sequent system where program closures are replaced by explicit rules handling left occurrences of logical constants.

Structural Rules

Proof rules **(I)-(VII)** assume implicitly the following structural rules, which are stated explicitly in other sequent systems:

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

Proof Rules as Search Operations

Proof rules **(I)-(VII)** implement an abstract interpreter for *hhl* in terms of the following search operations, which can be viewed as being directly determined by the proof rules themselves:

SUCCESS

If we are trying to find a proof for an atom A from a program $\mathcal{P}$, then such a proof terminates successfully if we can use proof rule **(I)**.

BACKCHAIN

Backchaining in the course of a proof is obtained by using proof rule **(II)** when, in trying to prove A, an instance of a definite clause of the form $G \supset A$ is in $[\mathcal{P}]$ and there is a proof of G from $\mathcal{P}$

AND

AND-nodes are obtained in a proof through the use of proof rule **(III)**, which derives a goal of the form $G_1 \wedge G_2$ from a program $\mathcal{P}$ when both G_1 and G_2 can be derived from $\mathcal{P}$

OR

Analogously, *OR*-nodes are obtained in a proof through the use of proof rule **(IV)**, which derives a goal of the form $G_1 \vee G_2$ from a program $\mathcal{P}$ when at least one of G_1 and G_2 can be derived from $\mathcal{P}$

INSTANCE

By proof rule **(V)**, to prove that an existentially quantified goal of the form $\exists x G$ can be derived from a program $\mathcal{P}$ we must be able to prove that an instance of G can be derived from $\mathcal{P}$

GENERIC

By proof rule **(VI)**, to prove that a universally quantified goal of the form $\forall x G$ can be derived from a program $\mathcal{P}$, we must be able to prove that $[x/c]G$ can be derived from $\mathcal{P}$, where c is a new parameter

AUGMENT

By proof rule **(VII)**, to prove that an implicational goal of the form $D \supset G$ can be derived from a program $\mathcal{P}$, we must be able to prove that G can be derived from the program obtained by augmenting $\mathcal{P}$ with D

Notice that, aside of *GENERIC* and *AUGMENT*, all of these search operations are also used in executing Prolog programs. Indeed, it is possible to view proof rules **(I)**-**(V)** as defining a sequent version of Prolog theorem proving. In fact, proof rules **(I)**-**(II)** formalize the backchaining strategy in the execution of Prolog programs; proof rule **(III)** accounts for the organization of the search space of a Prolog interpreter in terms of *AND*-nodes; proof rule **(IV)** accounts for *OR*-nodes, which are implicit in the non-determinism of Prolog interpreters, and can be made explicit in certain implementations of Prolog through the use of a disjunctive operator; finally, the *INSTANCE* operation, formalized through proof rule **(V)**, correspond to the possibility of instantiating existential variables in the course of a Prolog proof, which accounts for the procedural ability of Prolog to return values as output of the computation.

Observe also that the proof-theoretic account of universally quantified goals associated with the *GENERIC* search operation provides an *intensional* reading of universal quantification which is in a way complementary with the *extensional* reading associated with the model-theoretic account of universally quantified

$$\dfrac{\dfrac{\dfrac{\dfrac{\mathcal{P} \cup \{(P\ \ c)\} \Rightarrow (P\ \ c)}{\mathcal{P} \cup \{(P\ \ c)\} \Rightarrow (Q\ \ c)}\ \textbf{(II)} - \textit{BACKCHAIN}}{\mathcal{P} \cup \{(P\ \ c)\} \Rightarrow \exists v(Q\ \ v)}\ \textbf{(V)} - \textit{INSTANCE}}{\mathcal{P} \Rightarrow (P\ \ c) \supset \exists v(Q\ \ v)}\ \textbf{(VII)} - \textit{AUGMENT}}{\mathcal{P} \Rightarrow \forall z[(Q\ \ z) \supset \exists v(Q\ \ v)]}\ \textbf{(VI)} - \textit{GENERIC}$$

Figure 2.1: Proof tree for $\{\forall x[(P\ \ x) \supset (Q\ \ x)]\} \Rightarrow \forall z[(P\ \ z) \supset \exists v(Q\ \ v)]$

goals. Indeed, the latter account is justified by the semantics of universal quantification in the following way: given a program $\mathcal{P}$, a goal $\forall x G$ is true of $\mathcal{P}$ if for all terms t $[x/t]G$ is true of $\mathcal{P}$. This interpretation of universal quantification is often used in database applications and is implementable in Prolog systems using metalogical operators. See [26] for a formal treatment of this interpretation of universally quantified goals.

Throughout this thesis, we shall make an essential use of the *GENERIC* and of the *AUGMENT* search operations to account for situations which go beyond the expressive power of Prolog programs. Here below, we give two examples of the role played by such search operations in the course of *hhl* proofs.

A Simple Proof

Let $\mathcal{P}$ be $\{\forall x[(P\ \ x) \supset (Q\ \ x)]\}$. Figure 1.1 contains a proof tree for $\mathcal{P} \Rightarrow \forall z[(P\ \ z) \supset \exists v(Q\ \ v)]$. As this tree does not contain any branching produced by *AND*-nodes or *OR*-nodes, its size is identical to its depth, which is 5. The tree is here pictured as growing upward from its root

An Application to Knowledge Representation

The other example we consider here gives a direct application of *hhl* to the well-known "sterile jar" problem from knowledge representation. This example is particularly interesting because it involves a form of hypothetical reasoning in all respects similar to the one that will characterize our proof-theoretic reconstruction of the treatment of extraction phenomena in linguistic theories like GPSG and CG. The problem can be stated as follows: assume that a jar is

sterile if every germ in it is dead, that a germ in a heated jar is dead, and that a given jar has been heated. What reasoning is necessary to establish that the given jar is sterile? The intensional interpretation of universal quantification will work here. In fact, we can encode the three conditions above in a program $\mathcal{P}$ containing the following definite clauses:

$$\forall x[\forall y[(GERM\ y) \supset (IN\ x\ y) \supset (DEAD\ y)] \supset (STERILE\ x)]$$

$$\forall y \forall x[(HEATED\ x) \wedge (IN\ x\ y) \wedge (GERM\ y) \supset (DEAD\ y)]$$

$$(HEATED\ \mathbf{j})$$

Suppose now we want to prove the goal

$$(STERILE\ \mathbf{j})$$

The proof of such a goal can be informally described as follows. Backchaining on the first clause yields the goal

$$\forall y[(GERM\ y) \supset (IN\ \mathbf{j}\ y) \supset (DEAD\ \mathbf{j})]$$

Given the intensional interpretation of universal quantification, we proceed by selecting a "generic" parameter c, which does not occur in $\mathcal{P}$ or this goal. We now attempt to prove the goal

$$(GERM\ c) \supset (IN\ \mathbf{j}\ c) \supset (DEAD\ c)$$

This goal succeeds if the goal $(DEAD\ c)$ follows from the augmented program $\mathcal{P} \cup (GERM\ c) \cup (IN\ \mathbf{j}\ c)$. This indeed follows in two backchaining steps.

2.2.5 Logical Variables

In the logic programming terminology, a logical variable stays for a yet unspecified but completely unique term which is going to be specified in the course of the computation via unification. Procedurally, the accessing of a quantified

program clause, as a consequence of a backchaining step, is typically responsible for the introduction of logical variables which will instantiate the universally quantified variables of the clause. Analogously, the elimination of an existential quantifier in a goal clause via proof rule **(V)** will introduce a logical variable to instantiate an existentially quantified variable via the *INSTANCE* operation. In contrast to Horn Logic, in *hhl* logical variables can enter program space, because subparts of goal clauses can become program clauses via the *AUGMENT* operation. For example, consider a goal of the for $\exists x[(P \;\; x) \supset (Q \;\; x)]$, i.e. an existentially quantified implication in which the quantified variable occurs on both sides of the implication. If this quantifier is eliminated via proof rule **(V)** and the quantified variable x is replaced by a logical variable x^σ,then the left part $(P \;\; x^\sigma)$ must be added to the program before attempting to prove the right part $(Q \;\; x^\sigma)$. When unification provides substitutions for x^σ, then both program clauses and goals must be updated accordingly.

Since, whenever ambiguity does not arise, it is notationally convenient to avoid the use of explicit quantifiers, we shall also follow systematically the convention of suffixing logical variables with the symbol σ as a superscript in order to distinguish them from implicitly or explicitly quantified variables. So, for example, we can distinguish between a goal clause

$$(P \;\; x) \supset (Q \;\; x)$$

from where we have omitted the existential quantifier and its instantiation

$$(P \;\; x^\sigma) \supset (Q \;\; x^\sigma)$$

2.2.6 Quantifier Scoping

On the other hand, in *hhl* it is sometimes crucial to make use of explicit quantifiers in order to distinguish between different possibilities of quantifier scopings, which are here quite more complex than in Horn Clause Logic, where one can simply assume that all variables in program clauses are quantified by outermost universal quantifiers and all variables in goal clauses are quantified by outermost existential quantifiers. We illustrate here two examples where differ-

ent quantifier scopings crucially change the proof-theoretic meaning of a given formula.

First, consider the sequents

$$\Rightarrow \forall x[(P\ \ x) \supset (Q\ \ x)] \supset (((P\ \ a) \wedge (P\ \ b)) \supset ((Q\ \ a) \wedge (Q\ \ b)))$$

and

$$\Rightarrow \exists x[((P\ \ x) \supset (Q\ \ x)) \supset (((P\ \ a) \wedge (P\ \ b)) \supset ((Q\ \ a) \wedge (Q\ \ b)))]$$

Both these sequents are characterized by *hhl* implicational goal clauses, and are differing from each other only in the fact that in the first sequent the variable x is bound by a universal quantifier scoping over the antecedent of the implication, while in the second sequent the variable x is bound by an existential quantifier scoping over the whole implication. Clearly, the first sequent has a proof in terms of proof rules **(I)-(VII)**, while the second one does not. In fact, in the first case the *AUGMENT* operation will add to the (initially empty) program the universally quantified definite clause

$$\forall x[(P\ \ x) \supset (Q\ \ x)]$$

which can be instantiated an arbitrary number of times, so that it is possible to create subproofs both for $(Q\ \ a)$ and for $(Q\ \ b)$. By contrast, in the second sequent we first eliminate the existential quantifier via the *INSTANCE* operation, and then we add the clause

$$(P\ \ x^{\sigma}) \supset (Q\ \ x^{\sigma})$$

to the program; such a clause is however characterized by an occurrence of a logical variable, which cannot be unified both with the ground term a and with the ground term b. Therefore, with the second sequent it is not possible to have subproofs both for $(Q\ \ a)$ and $(Q\ \ b)$ (since suceeding with both of such proofs would imply to have two different unifications for x^{σ}), and so the whole goal fails.

For another example, consider now the sequents

$$\Rightarrow \forall x[(P\ \ x) \supset (Q\ \ x)] \supset \forall z[(P\ \ z) \supset \exists y(Q\ \ y)]$$

and

$$\Rightarrow \exists x[((P\ \ x) \supset (Q\ \ x)) \supset \forall z[(P\ \ z) \supset \exists y(Q\ \ y)]]$$

Again, the goal clauses in the two sequents differ only in the fact that the variable x in the first one is bound by a universal quantifier scoping over the antecedent of the implication, while in the second one the same variable is bound by an existential quantifier scoping over the whole clause. Again, only the first sequent has a proof. To see what is going on, consider that, as before, the *AUGMENT* operation will in the first case add to the program the universally quantified clause

$$\forall x[(P\ \ x) \supset (Q\ \ x)]$$

, and in the second case the clause characterized by the occurrence of a logical variable

$$(P\ \ x^{\sigma}) \supset (Q\ \ x^{\sigma}$$

We then proceed to select a generic parameter for the universally quantified goal

$$\forall z[(P\ \ z) \supset \exists y(Q\ \ y)]$$

, according to the *GENERIC* operation implemented in proof rule **(VII)**. But then only the proof of the first sequent can succeed, since in the second case there is no way of selecting a generic parameter which does violate the condition in proof rule **(VI)**, stating that the introduced generic must not occur free in the lower sequent of the inference figure associated with the rule itself. (This constraint on the use of generic parameters is known in the proof-theoretic literature as the *eigenvariable* condition.) In fact, for the proof to go through, we would need to unify the logical variable x^{σ} with the introduced generic, thus violating the *eigenvariable* condition.

We shall see that, from the point of view of grammatical formalisms, this possibility of differentiating between quantifier scopings provides a very powerful tool to enforce linguistic constraints in a declarative manner.

2.2.7 The Existential Property

As shown in [34], both Prolog programs and *hhl* programs are characterized by the *existential property*, that is the property that if $\mathcal{P} \Rightarrow \exists x G$ is provable, then there must be some term t such that $\mathcal{P} \Rightarrow [x/t]G$ is also provable. The existential property guarantees the ability of a given logic programming language to return definite answer substitution as a result of proving the derivability of an existential goal. Miller et al. [34] characterize this property as a direct consequence of the fact that the metatheory of Prolog and of *hhl* is Intuitionistic Logic, rather than Classical Logic. In fact, they show that, for any program $\mathcal{P}$ and goal G written in standard Horn logic syntax or in the syntax of *hhl*, $\mathcal{P} \Rightarrow G$ is provable in terms of proof rules **(I)-(VII)** if and only if $\mathcal{P} \Rightarrow G$ is intuitionistically provable.

Notice that viewing Prolog programs in terms of Classical Logic still grants definite answer substitutions, and indeed this is why Horn clause resolution is compatible with the existential property. However, once we extend standard Horn logic to *hhl*, classical provability does not grant the existential property anymore. For instance, consider the following sequent:

$$\{(P\ \ a) \wedge (P\ \ b) \supset Q\} \Rightarrow \exists x[(P\ \ x) \supset Q]$$

This sequent is not provable in Hereditary Harrop Logic, nor in Intuitionistic Logic, but can be derived in Classical Logic, as shown in [29], even though there is no definite answer substitution for the existential variable in the goal.

2.3 Intuitionistic versus Classical Logic

The language of full-blown Intuitionistic Logic is itself a proper superset of *hhl*. Indeed, in such a language it is possible to write sequents which are not expressible in *hhl* and which, as pointed out in [34], do not respect the search-related characterization of logical connectives given above. Thus, Hereditary Harrop Logic corresponds to the fragment of Intuitionistic Logic, with respect to which it is sound and complete, where logical connectives can be characterized in

terms of abstract search operations directly compatible with the goal-oriented paradigm around which has been built logic programming. Hereditary Harrop Logic is instead incomplete with respect to Classical Logic, since there are expressions, like the sequent above, which are both in *hhl* and in the language of Classical Logic, and which are classically provable but cannot be derived in terms of proof rules **(I)-(VII)**.

Since definite answer substitutions are an important and desirable property of logic programming languages, the use of a constructive logic like Intuitionistic Logic as the metatheory for logic programming shows that it is possible to go beyond Prolog while maintaining one of its most important features. From the point of view of natural language analysis, we shall show how this translates most usefully into a greater expressive power, with no loss of the capability of efficiently computing of definite analyses for given input strings.

Chapter 3

Definite Clause Grammars and Generalized Phrase Structure Grammars

3.1 Definite Clause Grammars

Definite Clause Grammars (DCGs) were introduced by Pereira and Warren [43], their direct ancestry being traceable to Colmerauer's more complicated framework of Metamorphosis Grammars [5]. The basic insight behind DCGs is that grammatical formalisms encoded as rewrite systems can be translated into sets of definite clauses. Each non-terminal symbol in the original grammar corresponds in the DCG notation to a predicate taking as arguments a certain number of *string positions*, plus other optional arguments. Parsing can then be viewed as theorem proving, and can be directly taken care of by the execution process of a logic programming language.

3.1.1 Definite Clause Grammars and Phrase Structure Grammars

An immediate and well-known application of DCGs is in translating phrase structure grammars into logical notation. This is also an application which is of particular interest to us here, since Generalized Phrase Structure Grammar

(GPSG) is itself a variation of phrase structure grammar, and the main content of this chapter is in showing how to extend the DCG framework in terms of Hereditary Harrop Logic so as to accomodate some of the features of GPSG.

Let us consider the simple phrase structure grammar in Figure 3.1 [1]. We can translate this grammar in definite clause notation as in Figure 3.2, by mapping its non-terminals into two-place predicates taking as arguments string positions. Strings are encoded as lists and string positions are represented in terms of the portion of the list they identify and of the substring which follows it, according to the familiar "difference-list" notation. We assume the use of the list constructor *CONS*, but we also interchangeably use the Prolog square bracket notation [...] for list representation. Thus, we shall read $(CONS\ \ x\ \ l)$ as being equivalent to Prolog infix notation $[x|l]$. We also follow the usual Prolog convention of taking unquantified variables in definite clauses to be implicitly universally quantified, with universal quantifiers scoping over the whole clause.

Adding Extra Arguments

We can add more information to the non-terminal predicates of a DCG by allowing extra arguments beside those representing string positions. For instance, suppose we want to distinguish the *number* of noun and verb-phrases - whether they are singular or plural - so as to guarantee that sentences are composed of noun-phrases and verb-phrases agreeing in number. This can be achieved by adding an additional argument to certain predicates, as in the DCG in Figure 3.3. Figure 3.4 shows a proof tree obtained from this DCG for the sentence *Paul loves Kay*, in terms of the proof rules of Hereditary Harrop Logic, where $\mathcal{P}$

[1] Throughout this thesis, we shall adopt the following more or less standard conventions for labels of syntactic categories: *S* stands for the category of sentences, *S_BAR* for that of *complement* clauses, e.g. sentences prefixed by the complementizer **that**, and *REL* for that of relative clauses; *NP* stands for the category of noun phrases, *PN* for that of proper names, *N* for that of nouns, and *DET* for that of determiners; *VP* stands for the category of verb phrases, *ADVP* for that of verb phrase modifiers, *TV* for that of transitive verbs, and *STV* for the category of verbs taking as arguments sentence complements; finally, *PREP* and *PP* stand, respectively, for the categories of prepositional phrases and of prepositions.

$$
\begin{array}{lll}
S & \rightarrow & NP \quad VP \\
VP & \rightarrow & TV \quad NP \\
VP & \rightarrow & STV \quad S\_BAR \\
S\_BAR & \rightarrow & \textbf{that} \quad S \\
NP & \rightarrow & DET \quad N \\
NP & \rightarrow & DET \quad N \quad PP \\
NP & \rightarrow & PN \\
PP & \rightarrow & PREP \quad NP \\
DET & \rightarrow & \textbf{the} \\
N & \rightarrow & \textbf{sister} \\
N & \rightarrow & \textbf{woman} \\
PN & \rightarrow & \textbf{Kay} \\
PN & \rightarrow & \textbf{Fred} \\
PN & \rightarrow & \textbf{Paul} \\
TV & \rightarrow & \textbf{loves} \\
TV & \rightarrow & \textbf{married} \\
STV & \rightarrow & \textbf{believes} \\
PREP & \rightarrow & \textbf{of}
\end{array}
$$

Figure 3.1: Example of phrase structure grammar

$$(NP\ x\ z) \wedge (VP\ z\ y) \supset (S\ x\ y)$$

$$(TV\ x\ z) \wedge (NP\ z\ y) \supset (VP\ x\ y)$$

$$(STV\ x\ z) \wedge (S\_BAR\ z\ y) \supset (VP\ x\ y)$$

$$(S\ x\ y) \supset (S\_BAR\ (CONS\ \textbf{that}\ x)\ y)$$

$$(DET\ x\ z) \wedge (N\ z\ y) \supset (NP\ x\ y)$$

$$(DET\ x\ z) \wedge (N\ z\ y) \wedge (PP\ y\ v) \supset (NP\ x\ v)$$

$$(PN\ x\ y) \supset (NP\ x\ y)$$

$$(PREP\ x\ z) \wedge (NP\ z\ y) \supset (PP\ x\ y)$$

$$(DET\ (CONS\ \textbf{the}\ l)\ l)$$

$$(N\ (CONS\ \textbf{sister}\ l)\ l)$$

$$(N\ (CONS\ \textbf{woman}\ l)\ l)$$

$$(PN\ (CONS\ \textbf{Kay}\ l)\ l)$$

$$(PN\ (CONS\ \textbf{Fred}\ l)\ l)$$

$$(PN\ (CONS\ \textbf{Paul}\ l)\ l)$$

$$(TV\ (CONS\ \textbf{loves}\ l)\ l)$$

$$(TV\ (CONS\ \textbf{married}\ l)\ l)$$

$$(STV\ (CONS\ \textbf{believes}\ l)\ l)$$

$$(PREP\ (CONS\ \textbf{of}\ l)\ l)$$

Figure 3.2: A DCG encoding the phrase structure grammar in Figure 3.1

is the program given by the grammar in Figure 3.3. For the sake of readability, string arguments have been omitted from the nodes in the trees, and agreement arguments are the only one displayed.

Here below, in our augmentation of DCGs via GPSG-style non-terminals, we shall make use of an extra argument, beside the ones corresponding to string positions, to construct a logical form for the string being parsed. We shall exploit for this purpose the use of β-reduction provided by unification with built-in β-reduction.

3.2 Syntactic Categories in GPSG

The GKPS book [10] on GPSG is an impressive attempt to give a thorough formalization of a system for the grammatical description of natural language. Many aspects of such a formalization are beyond our scope here. Indeed, our purpose is that of extrapolating some of the concepts formalized in GKPS and show how they can be realized more or less differently within DCGs based on the *hhl* extension of Horn Logic.

Our main target is the notion of syntactic category which is obtained through the GPSG approach to phrase structure grammar. (For a recent discussion on this subject, see also [11].). Particularly relevant are the following aspects:

(i) A GPSG category augments the bare non-terminals of simple phrase structure grammars with *morpho-syntactic* information, i. e. information concerning part of speech, inflection, case, agreement etc. Such information is encoded in terms of *features*, i.e. attribute-value pairs of the form [number sg].

(ii) GPSG categories are allowed in slashed form, that is in the form X/Y, the intuitive meaning being that we have a constituent of category X with an internal gap (i.e., a missing constituent) of category Y. In this way GPSG can elegantly express filler-gap (unbounded) dependencies; for instance, a possible rule for relative clauses can be stated as

$$REL \rightarrow \textbf{whom} \quad S/NP$$

$(NP\ x\ z\ num) \wedge (VP\ z\ y\ num) \supset (S\ x\ y)$

$(TV\ x\ z\ num) \wedge (NP\ z\ y\ num_1) \supset (VP\ x\ y\ num)$

$(STV\ x\ z\ num) \wedge (S\_BAR\ z\ y) \supset (VP\ x\ y\ num)$

$(S\ x\ y) \supset (S\_BAR\ (CONS\ \textbf{that}\ x)\ y)$

$(DET\ x\ z\ num) \wedge (N\ z\ y\ num) \supset (NP\ x\ y\ num)$

$(DET\ x\ z\ num) \wedge (N\ z\ y\ num) \wedge (PP\ y\ v) \supset (NP\ x\ v\ num)$

$(PN\ x\ y\ num) \supset (NP\ x\ y\ num)$

$(PREP\ x\ z) \wedge (NP\ z\ y\ num) \supset (PP\ x\ y)$

$(DET\ (CONS\ \textbf{the}\ l)\ l\ num)$

$(NP\ (CONS\ \textbf{men}\ l)\ l\ \textbf{pl})$

$(N\ (CONS\ \textbf{sister}\ l)\ l\ \textbf{sg})$

$(N\ (CONS\ \textbf{woman}\ l)\ l\ \textbf{sg})$

$(PN\ (CONS\ \textbf{Kay}\ l)\ l\ \textbf{sg})$

$(PN\ (CONS\ \textbf{Fred}\ l)\ l\ \textbf{sg})$

$(PN\ (CONS\ \textbf{Paul}\ l)\ l\ \textbf{sg})$

$(TV\ (CONS\ \textbf{loves}\ l)\ l\ \textbf{sg})$

$(TV\ (CONS\ \textbf{married}\ l)\ l\ num)$

$(STV\ (CONS\ \textbf{believes}\ l)\ l\ \textbf{sg})$

$(PREP\ (CONS\ \textbf{of}\ l)\ l)$

Figure 3.3: DCG encoding a phrase structure grammar augmented with agreement information

$$\dfrac{\dfrac{\mathcal{P} \Rightarrow (PN\ \textbf{sg})}{\mathcal{P} \Rightarrow (NP\ \textbf{sg})}\,\textbf{(II)} \qquad \dfrac{\mathcal{P} \Rightarrow (TV\ \textbf{sg}) \qquad \dfrac{\mathcal{P} \Rightarrow (PN\ \textbf{sg})}{\mathcal{P} \Rightarrow (NP\ \textbf{sg})}\,\textbf{(II)}}{\mathcal{P} \Rightarrow (TV\ \textbf{sg}) \wedge (NP\ \textbf{sg})}\,\textbf{(III)}}{\dfrac{\mathcal{P} \Rightarrow (VP\ \textbf{sg})}{\mathcal{P} \Rightarrow (NP\ \textbf{sg}) \wedge (VP\ \textbf{sg})}\,\textbf{(III)}}\,\textbf{(II)}$$

Figure 3.4: Proof tree for **Paul loves Kay**

(iii) GPSG states explicitly how to build the logical form for a given string via rules of semantic interpretation which come in pairs with the syntactic rules. Such semantic rules are inspired by Montague's principle of compositionality [35], and view the interpretation of a sentence as obtained from the combination of the interpretations of its subconstituents, where the method of combination is given by functional application and β-contraction. Thus, the rule in *(ii)* can be paired with a rule of semantic interpretation as follows:

$$REL \rightarrow \textbf{whom} \quad S/NP \qquad S/NP'$$

(The prime notation " ′ " refers here to the semantic counterpart of a given syntactic category.) This pairing provides the information that the semantic interpretation of a relative clause is given by the semantic interpretation of the sentence where the gap occurs.

Now, it is well-known that an augmentation of phrase structure grammars of the kind described in *(i)* can be automatically implemented in DCGs by adding extra arguments as in the example in Figure 3.3. For this reason, we shall not be further concerned with it here. On the other hand, syntactic categories of the kind described in *(ii)* are also formalized in GKPS by assuming that the category corresponding to the missing constituent is a feature of the category where the gap occurs. However, in contrast to the morpho-syntactic features of *(i)*, category-valued features of this kind cannot be automatically handled through unification, and must obey the complicated principles of feature percolation stated in GKPS. We offer here a proof-theoretic alternative to this approach, where the category on the right of the slash is not viewed as a feature of the category on the left, but rather the whole slashed category is viewed as an implication, with the slash as the implication connective, and the category on the left and the one on the right as, respectively, the consequent and the antecedent in the implication. We show then that the process of parsing with this kind of grammars reduces to theorem proving with Hereditary Harrop Logic, without need of any further machinery. Moreover, we provide a natural and elegant implementation of *(iii)*, by embedding the rules of semantic interpretation

into their syntactic counterparts by passing logical forms as extra arguments of non-terminal predicates and exploiting the mechanism of β-reduction.

3.3 Hereditary Harrop Logic, Definite Clause Grammars, and Generalized Phrase Structure Grammars

We go now in the details of the task of translating the GPSG treatment of unbounded dependencies in the DCG framework by extending DCGs themselves in terms of Hereditary Harrop Logic. Again, it is worth repeating that the purpose of our enterprise does not consist in defining a procedure for compiling full-blown GPSGs into DCGs. Rather, the goal here is that of encorporating in the DCG framework a treatment of filler-gap dependencies which is directly reminiscent of the GPSG treatment of the same phenomena, and which is directly licensed by the inferential machinery of a novel general-purpose logic programming language. We can show in this way that an important application of such a language is in providing a well-defined methodology for dealing with crucially important problems of linguistic analysis.

However, the accomplishment of our enterprise can certainly throw some light on the different, and undoubtely more complex, task of providing a full logical reconstruction of GPSG, and it may even suggest alternatives to the formalization of GPSG given in GKPS. Indeed, the extended DCGs introduced in this chapter use a notation for gap-introducing rules and gap-eliminating rules closely related to the GPSG notation for the same kind of rules, and produce the same logical forms as semantic representations of well-formed sentences. On the other hand, what is in between - that is, the way in which filler-gap dependencies are resolved - differs largely from the approach taken in GKPS, where gaps are treated as "features", and are percolated through phrase structure trees in terms of elaborated graph-theoretical mechanisms of percolation. By contrast, our approach treats gaps simply as standard DCG non-terminals handled through the *AUGMENT* and *GENERIC* search operations of the *hhl*

interpreter. The suitability of this approach extends, as we shall see, to cases of "pied-piping" in filler-gap dependencies, which have been brought forward to justify the use of graph-theoretical principles of feature percolation for handling filler-gap dependencies. Immediate advantages of this proof-theoretic account of unbounded dependencies are its simplicity and efficiency, deriving from the fact that introduction and elimination of gaps is directly taken care of by the underlying logical engine. As we shall point out, another advantage is in the reduced size of the grammar with respect to the GKPS formalization.

From the point of view of the formalization of linguistic theories, perhaps the most significant conclusion which can be reached from our account of unbounded dependencies is that it provides a surprising counterargument to a point raised by Pollard [44] in the context of a discussion of the differences and similarities between GPSG and CG, the linguistic framework we are going to discuss in Chapter 5. Pollard's point is that linguistic evidence, in the guise of the infamous "pied-piping" cases, opposes the claim of certain Categorial Grammar practitioners that unbounded dependencies can be treated without using any mechanism of feature percolation; as a consequence, the gaps-as-features approach proposed for GPSG has to be imported in some way into Categorial Grammar. But what our approach suggests is that gaps-as-features can be eliminated altogether from GPSG itself! Our treatment of filler-gap dependencies will in fact show that GPSG and Categorial Grammar can be viewed as strikingly similar, with Categorial Grammar corresponding to a "lexicalized" version of GPSG. On the other hand, commitment to lexical knowledge (in the case of Categorial Grammar) and lack of such a commitment (in the case of GPSG) will cause subtle and far-reaching differences in the way in which rules are used during parsing, and constraints over certain syntactic constructions are implemented.

3.3.1 Rules not Covering Filler-gap Dependencies

The fundamental step in making sense in terms of DCGs of the GPSG treatment of filler-gap dependencies consists in finding a correspondence between GPSG

S	$\rightarrow$	$NP \quad VP$	$VP'(NP')$
VP	$\rightarrow$	$TV \quad NP$	$TV'(NP')$
VP	$\rightarrow$	$STV \quad S\_BAR$	$STV'(S\_BAR')$
S_BAR	$\rightarrow$	**that** S	S'
NP	$\rightarrow$	$DET \quad N$	$DET'(N')$
NP	$\rightarrow$	$DET \quad N \quad PP$	$PP'(DET'(N'))$
NP	$\rightarrow$	PN	PN'
PP	$\rightarrow$	$PREP \quad NP$	$PREP'(NP')$
DET	$\rightarrow$	**the**	**the**$'$
N	$\rightarrow$	**sister**	**sister**$'$
N	$\rightarrow$	**woman**	**woman**$'$
PN	$\rightarrow$	**Kay**	**Kay**$'$
PN	$\rightarrow$	**Fred**	**Fred**$'$
PN	$\rightarrow$	**Paul**	**Paul**$'$
TV	$\rightarrow$	**loves**	**love**$'$
TV	$\rightarrow$	**married**	**married**$'$
STV	$\rightarrow$	**believes**	**believe**$'$
$PREP$	$\rightarrow$	**of**	**of**$'$

Figure 3.5: Set of GPSG rules

rules and formulae of *hhl*. We start with the easier part of the task, consisting in finding a correspondence for rules where no filler-gap dependencies are involved. For this purpose, the power of Hereditary Harrop Logic is redundant, and standard Horn Logic is enough. Thus, consider the set of GPSG rules in Figure 3.5. The grammar they define allows the generation of phrase structure trees like the one in Figure 3.6. A corresponding set of definite clauses is given by the logic program $\mathcal{P}_1$ in Figure 3.7. The semantic part of the GPSG rules has here been passed there as an extra-argument to the DCG non-terminals.

These clauses fall within the Horn-clause subset of *hhl*, and, as such, do not involve any use of $GENERIC$ and $AUGMENT$ operations. However, notice that

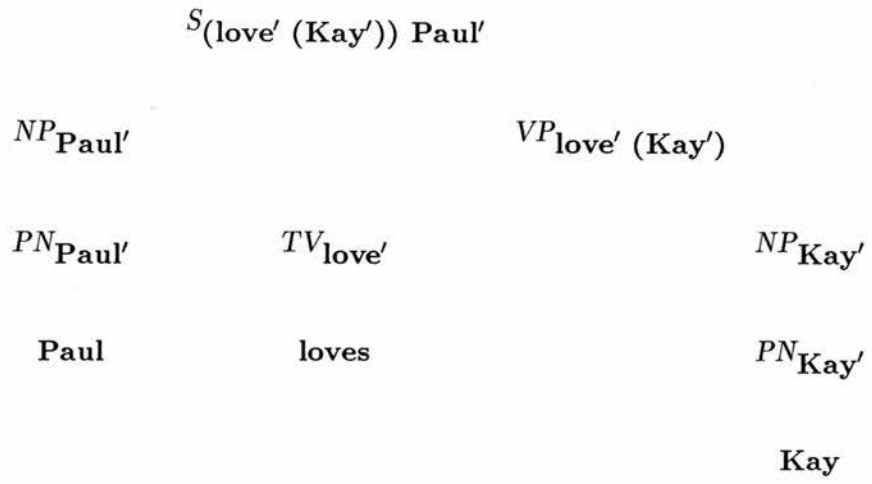

Figure 3.6: GPSG analysis for **Paul loves Kay**

quantification over functions allows a direct and elegant implementation of the semantics. A proof tree corresponding to the phrase structure tree in Figure 3.6 is given in Figure 3.8, where string arguments have been omitted for the sake of readability, and semantic arguments are the only ones displayed. The search operations involved in such a proof are *BACKCHAIN*, *AND* and *INSTANCE*, corresponding, respectively, to proof rules **(II)**, **(III)** and **(V)**.

3.3.2 Rules Covering Filler-gap Dependencies - I

Suppose now we want to account for sentences involving filler-gap dependencies, such as

Fred loves the woman whom [Kay believes that Paul married $\uparrow$]

(We indicate the position of the gap with an upward-looking arrow.)

According to the gaps-as-features version of GPSG, this would involve:

(i) introducing the gap in terms of the following rule, which would need to be added to the rules of the previous section:

$$REL \rightarrow \textbf{whom} \quad S/NP \qquad S/NP'$$

$(NP\ x\ z\ np) \wedge (VP\ z\ y\ vp) \supset (S\ x\ y\ (vp\ np))$

$(TV\ x\ z\ tv) \wedge (NP\ z\ y\ np) \supset (VP\ x\ y\ (tv\ np))$

$(STV\ x\ z\ stv) \wedge (S\_BAR\ z\ y\ s\_bar) \supset (VP\ x\ y\ (stv\ s\_bar))$

$(S\ x\ y\ s) \supset (S\_BAR\ (CONS\ \mathbf{that}\ x)\ y\ s)$

$(DET\ x\ z\ det) \wedge (N\ z\ y\ n) \supset (NP\ x\ y\ (det\ n))$

$(DET\ x\ z\ det) \wedge (N\ z\ y\ n) \wedge (PP\ y\ v\ pp) \supset (NP\ x\ v\ (pp\ (det\ n)))$

$(PREP\ x\ z\ prep) \wedge (NP\ z\ y\ np) \supset (PP\ x\ y\ (prep\ np))$

$(PN\ x\ y\ pn) \supset (NP\ x\ y\ pn)$

$(DET\ (CONS\ \mathbf{the}\ l)\ l\ \mathbf{the'})$

$(N\ (CONS\ \mathbf{sister}\ l)\ l\ \mathbf{sister'})$

$(N\ (CONS\ \mathbf{woman}\ l)\ l\ \mathbf{woman'})$

$(PN\ (CONS\ \mathbf{Kay}\ l)\ l\ \mathbf{Kay'})$

$(PN\ (CONS\ \mathbf{Fred}\ l)\ l\ \mathbf{Fred'})$

$(PN\ (CONS\ \mathbf{Paul}\ l)\ l\ \mathbf{Paul'})$

$(TV\ (CONS\ \mathbf{loves}\ l)\ l\ \mathbf{love'})$

$(TV\ (CONS\ (\mathbf{married}\ l)\ l\ \mathbf{married'})$

$(STV\ (CONS\ (\mathbf{believes}\ l)\ l\ \mathbf{believe'})$

$(PREP\ (CONS\ (\mathbf{of}\ l)\ l\ \mathbf{of'})$

Figure 3.7: *hhl* version of the GPSG rules in Figure 3.5

$$\dfrac{\dfrac{\dfrac{\dfrac{\mathcal{P}_1 \Rightarrow (PN\ \mathbf{Paul'})}{\mathcal{P}_1 \Rightarrow (NP\ \mathbf{Paul'})}\,\mathbf{(II)} \qquad \dfrac{\dfrac{\mathcal{P}_1 \Rightarrow (TV\ \mathbf{love'}) \qquad \dfrac{\mathcal{P}_1 \Rightarrow (PN\ \mathbf{Kay'})}{\mathcal{P}_1 \Rightarrow (NP\ \mathbf{Kay'})}\,\mathbf{(II)}}{\mathcal{P}_1 \Rightarrow (TV\ \mathbf{love'}) \wedge (NP\ (\mathbf{Kay'})}\,\mathbf{(III)}}{\mathcal{P}_1 \Rightarrow (VP\ (\mathbf{love'}\ \mathbf{Kay'}))}\,\mathbf{(II)}}{\mathcal{P}_1 \Rightarrow (NP\ \mathbf{Paul'}) \wedge (VP\ (\mathbf{love'}\ \mathbf{Kay'}))}\,\mathbf{(III)}}{\mathcal{P}_1 \Rightarrow (S\ ((\mathbf{love'}\ \mathbf{Kay'})\ \mathbf{Paul'}))}\,\mathbf{(II)}}{\mathcal{P}_1 \Rightarrow \exists s(Ss)}\,\mathbf{(V)}$$

Figure 3.8: Proof tree for **Paul loves Kay**

(ii) percolating the gap down the tree in terms of principles of feature percolation

(iii) finally, locating the gap in terms of the following gap-terminating rule

$$VP/NP \;\rightarrow\; TV \quad NP/NP \qquad \lambda np TV'(NP/NP'(np))$$

which is obtained as a generalization of the rule

$$VP \;\rightarrow\; TV \quad NP \qquad TV'(NP')$$

via an application of a metarule for *slash termination.* (GPSG metarules play the role of mapping an initial grammar into an expanded grammar, thus allowing grammatical generalizations across different sets of rules.)

For *(i)-(iii)* to work, we have also to add to the grammar the following null transition, where the empty string will correspond in the derivation to the gap itself:

$$NP/NP \;\rightarrow\; \epsilon \quad \lambda np\, np$$

Thus, under this approach, the semantic representation of a relative clause corresponds to a complex predicate encoded as a λ-expression with the gap individuating a λ-parameter. This machinery allows the derivation of phrase structure trees like the one in Figure 3.9 for the relative clause *whom Kay believes that Paul married.*

A Proof-theoretic View of Filler-gap Dependencies

Here below, we show how to encode a rule like the one in *(i)* in a corresponding definite clause of *hhl*, characterized by an internal implication and an internal universal quantifier. We then completely eliminate the need for step *(ii)*, by replacing feature percolation of the gap with proof-theoretic operations which are part of the overall inference mechanism behind Hereditary Harrop Logic; thus, filler-gap dependencies do not have here special status with respect to the other parts of the grammar. (As just mentioned earlier, and as will be

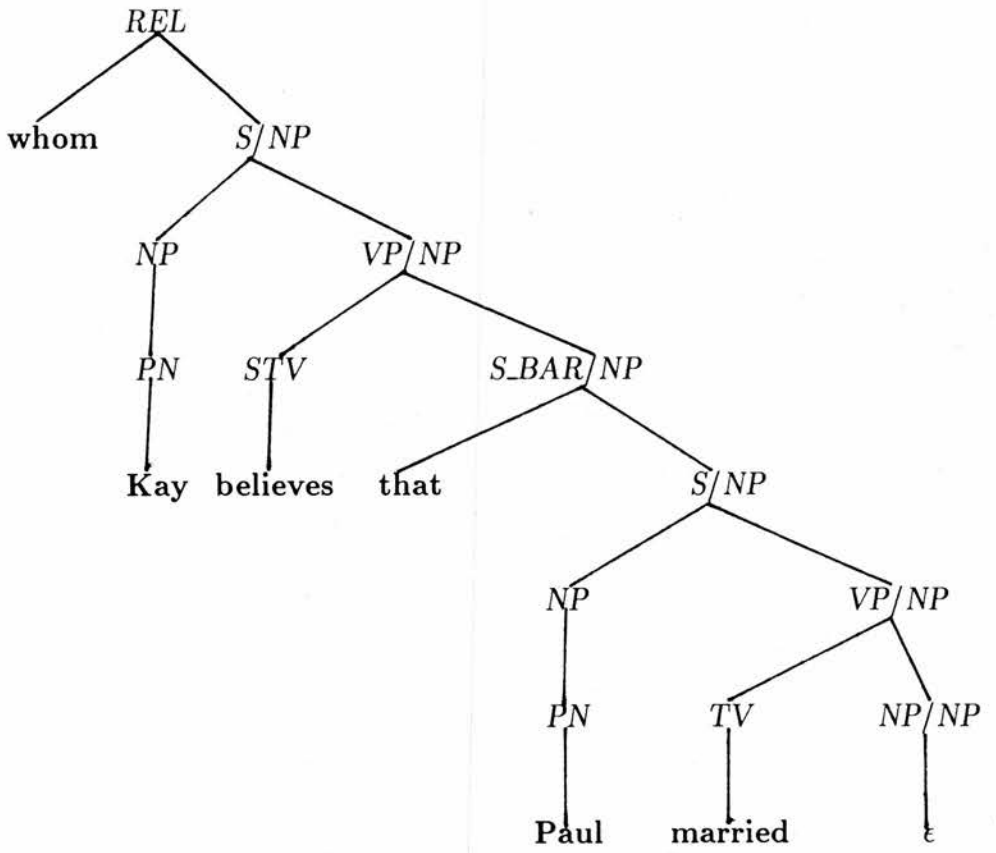

Figure 3.9: GPSG analysis for **whom Kay believes that Paul married**

further articulated later, this treatment of filler-gap dependencies embodies a "Categorial Grammar" view of GPSG.)

Since we want in the first place to give an intuitive flavour of our proof-theoretic approach to gap-introduction, we shall for the time being ignore step *(iii)*, and return to it in Section 3.3.4 below. In our case, the use of specific rules to "terminate" the gap is not as crucial as in the feature-based approach to gaps; in fact, we are able to obtain a proof tree analogous to the phrase structure tree in figure 3.9 without resorting to gap-terminating rules. On the other hand, gap-terminating rules will ultimately be needed in order to constrain the distribution of the gaps. However, we shall show that in our framework step *(i)* and step *(iii)* can be completely collapsed together. Another important and, as we shall point out, advantageous feature of our approach is that it completely avoids null transitions. In fact, empty constituents corresponding to gaps are just temporarily introduced in the course of the parsing process via the combined use of the *AUGMENT* and *GENERIC* search operations.

Gap Introduction as Hypothesis Introduction

We provide a definite clause version of the rule in *(i)* simply by interpreting the slash as implication, and putting a universal quantifier, which will then involve the introduction of a generic, in the argument for the hypothetical noun-phrase corresponding to the gap. Thus, we have the following definite clause:

$$\forall np[(NP\ z\ z\ np) \supset (S\ x\ y\ (rel\ np))] \supset (REL\ (CONS\ \textbf{whom}\ x)\ y\ rel)$$

Notice that we do not need to explicitly constrain the position of the hypothetical noun-phrase to be inbetween the string boundaries of the sentence where the gap occurs; such a constraint is automatically obtained from the fact that the noun-phrase is introduced locally and temporarily just to prove the given sentence, and is removed afterwards.

Now, let $\mathcal{P}_2$ be the program $\mathcal{P}_1$ augmented by the definite clause above. Then we can parse the relative clause *whom Kay believes that Paul married*

in terms of the proof tree in figure 3.10. The gap is introduced via a step of hypothesis introduction, in terms of proof rule **(VII)**, corresponding to the *AUGMENT* search operation; the semantic representation associated with it is the generic variable c introduced by proof rule **(VI)** for universally quantified goals. The variable corresponding to the string position where the gap occurs is instead (implicitly) bound by a universal quantifier having as scope the whole definite clause where the hypothetical noun-phrase predicate occurs; therefore, such a hypothetical noun-phrase predicate can locate only one gap in the course of the proof, since when it is added to the program its string position arguments are instantiated by the logical variables created at the moment of accessing the larger definite clause for backchaining. The reader may want to contrast this situation with the one that would be obtained by using as gap-introducing rule the definite clause

$$\forall np[\forall z[(NP\ z\ z\ np)] \supset (S\ x\ y\ (rel\ np))] \supset$$
$$(REL\ (CONS\ \textbf{whom}\ x)\ y\ rel)$$

In fact, in this case the string position corresponding to the gap would be a universal variable having as scope the atomic program clause encoding the gap. Clearly, this would imply that more than one gap position could be located by adding such a definite clause to the program. Thus, the contrast between this "too liberal" gap-terminating rule and the former one is yet another instance, in addition to the examples we have considered in Section 2.2.6 of Chapter 2, of the different proof-theoretic behaviour induced by different scopings of quantifiers.

We analyze the semantic representation of the sentence containing the gap as the functional application of the semantic representation rel^{σ} of the relative clause to the generic c. (Following the convention introduced in the previous chapter, rel^{σ} refers here to the yet unspecified term - i.e. to the logical variable - which substitutes the bound variable *rel* in the course of the proof.) Computationally, this kind of analysis can be elegantly carried out via unification with built-in β-reduction; the relevant step of β-reduction is here explicitly indicated in the proof tree with the arrow $\rightarrow_{\beta}$. In other words, we "decompose"

$$\frac{\mathcal{P}_2 \cup \{(NP\ c)\} \Rightarrow (TV\ \mathbf{married'}) \qquad \mathcal{P}_2 \cup \{(NP\ c)\} \Rightarrow (NP\ c)}{\mathcal{P}_2 \cup \{(NP\ c)\} \Rightarrow (TV\ \mathbf{married'}) \wedge (NP\ c)}\ (\mathbf{III})$$

$$\frac{\mathcal{P}_2 \cup \{(NP\ c)\} \Rightarrow (TV\ \mathbf{married'}) \wedge (NP\ c)}{\mathcal{P}_2 \cup \{(NP\ c)\} \Rightarrow (VP\ (\mathbf{married'}\ c))}\ (\mathbf{II})$$

$$\frac{\mathcal{P}_2 \cup \{(NP\ c)\} \Rightarrow (PN\ \mathbf{Paul'})}{\mathcal{P}_2 \cup \{(NP\ c)\} \Rightarrow (NP\ \mathbf{Paul'})}\ (\mathbf{II})$$

$$\frac{\mathcal{P}_2 \cup \{(NP\ c)\} \Rightarrow (NP\ \mathbf{Paul'}) \qquad \mathcal{P}_2 \cup \{(NP\ c)\} \Rightarrow (VP\ (\mathbf{married'}\ c))}{\mathcal{P}_2 \cup \{(NP\ c)\} \Rightarrow (NP\ \mathbf{Paul'}) \wedge (VP\ (\mathbf{married'}\ c))}\ (\mathbf{III})$$

$$\frac{\mathcal{P}_2 \cup \{(NP\ c)\} \Rightarrow (NP\ \mathbf{Paul'}) \wedge (VP\ (\mathbf{married'}\ c))}{\mathcal{P}_2 \cup \{(NP\ c)\} \Rightarrow (S\ ((\mathbf{married'}\ c)\ \mathbf{Paul'}))}\ (\mathbf{II})$$

$$\frac{\mathcal{P}_2 \cup \{(NP\ c)\} \Rightarrow (S\ ((\mathbf{married'}\ c)\ \mathbf{Paul'}))}{\mathcal{P}_2 \cup \{(NP\ c)\} \Rightarrow (S\_BAR\ ((\mathbf{married'}\ c)\ \mathbf{Paul'}))}\ (\mathbf{II})$$

$$\frac{\mathcal{P}_2 \cup \{(NP\ c)\} \Rightarrow (STV\ \mathbf{believe'}) \qquad \mathcal{P}_2 \cup \{(NP\ c)\} \Rightarrow (S\_BAR\ ((\mathbf{married'}\ c)\ \mathbf{Paul'}))}{\mathcal{P}_2 \cup \{(NP\ c)\} \Rightarrow (STV\ \mathbf{believe'}) \wedge (S\_BAR\ ((\mathbf{married'}\ c)\ \mathbf{Paul'}))}\ (\mathbf{III})$$

$$\frac{\mathcal{P}_2 \cup \{(NP\ c)\} \Rightarrow (STV\ \mathbf{believe'}) \wedge (S\_BAR\ ((\mathbf{married'}\ c)\ \mathbf{Paul'}))}{\mathcal{P}_2 \cup \{(NP\ c)\} \Rightarrow (VP\ ((\mathbf{believe'}((\mathbf{married'}\ c)\ \mathbf{Paul'})))}\ (\mathbf{II})$$

$$\frac{\mathcal{P}_2 \cup \{(NP\ c)\} \Rightarrow (PN\ \mathbf{Kay'})}{\mathcal{P}_2 \cup \{(NP\ c)\} \Rightarrow (NP\ \mathbf{Kay'})}\ (\mathbf{II})$$

$$\frac{\mathcal{P}_2 \cup \{(NP\ c)\} \Rightarrow (NP\ \mathbf{Kay'}) \qquad \mathcal{P}_2 \cup \{(NP\ c)\} \Rightarrow (VP\ ((\mathbf{believe'}((\mathbf{married'}\ c)\ \mathbf{Paul'})))}{\mathcal{P}_2 \cup \{(NP\ c)\} \Rightarrow (NP\ \mathbf{Kay'}) \wedge (VP\ ((\mathbf{believe'}((\mathbf{married'}\ c)\ \mathbf{Paul'})))}\ (\mathbf{III})$$

$$\frac{\mathcal{P}_2 \cup \{(NP\ c)\} \Rightarrow (NP\ \mathbf{Kay'}) \wedge (VP\ ((\mathbf{believe'}((\mathbf{married'}\ c)\ \mathbf{Paul'})))}{\mathcal{P}_2 \cup \{(NP\ c)\} \Rightarrow (S\ ((\mathbf{believe'}((\mathbf{married'}\ c)\ \mathbf{Paul'}))\ \mathbf{Kay'}))}\ (\mathbf{II})$$

$$\frac{\mathcal{P}_2 \cup \{(NP\ c)\} \Rightarrow (S\ ((\mathbf{believe'}((\mathbf{married'}\ c)\ \mathbf{Paul'}))\ \mathbf{Kay'}))}{\mathcal{P}_2 \Rightarrow (NP\ c) \supset (S\ [(\lambda np((\mathbf{believe'}((\mathbf{married'}\ np)\ \mathbf{Paul'}))\ \mathbf{Kay'})\ c) \rightarrow_\beta ((\mathbf{believe'}((\mathbf{married'}\ c)\ \mathbf{Paul'}))\ \mathbf{Kay'})])}\ (\mathbf{VII})$$

$$\frac{\mathcal{P}_2 \Rightarrow (NP\ c) \supset (S\ [(\lambda np((\mathbf{believe'}((\mathbf{married'}\ np)\ \mathbf{Paul'}))\ \mathbf{Kay'})\ c) \rightarrow_\beta ((\mathbf{believe'}((\mathbf{married'}\ c)\ \mathbf{Paul'}))\ \mathbf{Kay'})])}{\mathcal{P}_2 \Rightarrow \forall np[(NP\ np) \supset (S\ (\lambda np((\mathbf{believe'}((\mathbf{married'}\ np)\ \mathbf{Paul'}))\ \mathbf{Kay'})\ np))]}\ (\mathbf{VI})$$

$$\frac{\mathcal{P}_2 \Rightarrow \forall np[(NP\ np) \supset (S\ (\lambda np((\mathbf{believe'}((\mathbf{married'}\ np)\ \mathbf{Paul'}))\ \mathbf{Kay'})\ np))]}{\mathcal{P}_2 \Rightarrow (REL\ (\lambda np((\mathbf{believe'}((\mathbf{married'}\ np)\ \mathbf{Paul'}))\ \mathbf{Kay'})))}\ (\mathbf{II})$$

Figure 3.10: Proof tree for **whom Kay believes that Paul married**

the representation of the sentence containing the gap in terms of the unification problem

$$(rel^{\sigma}\ \ c) = ((\mathbf{believe'}((\mathbf{marry'}\ c)\ \mathbf{Paul'}))\ \mathbf{Kay'})$$

where

$$((\mathbf{believe'}((\mathbf{marry'}\ c)\ \mathbf{Paul'}))\ \mathbf{Kay'})$$

is the semantic interpretation of the sentence itself. The only possible solution for such a unification problem, in the context of the proof contained in figure 3.10, binds rel^{σ} to the λ-expression

$$\lambda np((\mathbf{believe'}((\mathbf{marry'}\ np)\ \mathbf{Paul'}))\ \mathbf{Kay'})$$

Clearly, this λ-expression provides us with the desired semantic representation for the relative clause.

Notice that unification with built-in β-reduction is in general a more complex task than the simple first-order unification used in ordinary Prolog, and may admit more than one unifier as a solution for a given unification problem [19]. So, for instance, in principle the unification problem above admits also the "vacuous" solution which binds rel^{σ} to

$$\lambda np((\mathbf{believe'}((\mathbf{marry'}\ c)\ \mathbf{Paul'}))\ \mathbf{Kay'})$$

On the other hand, this vacuous solution is, in the context of the proof in figure 3.10, automatically ruled out by the fact that it would violate the *eigenvariable* condition imposed by the use of generics; in fact, such a solution would imply that the generic c occurs free inside the term rel^{σ}, which is contained in the clauses involved in the step of the proof where rule **(VI)** is applied and c is introduced. However, as we shall see, not all undesired cases of vacuous abstraction can be automatically ruled out via the simple interaction of the components of *hhl* as in this case.

Some Considerations on Semantic Interpretation

There are several considerations which can be made on the approach to semantic interpretation we have taken above. In first place, there is a direct relationship

between this approach and the idea that formulae can be viewed as types of λ-terms, and that the corresponding λ-terms encode the proof of the formulae which are assigned to them as types. In fact, we can view the semantic representation associated with a parsed string as a way of encoding its own proof; or, alternatively, we can think of the syntactic category associated with a parsed string as the type of the semantic representation itself. This relationship between formulae and types, known in proof theory as the *Curry-Howard* isomorphism [18], has already been exploited in other applications of *hhl*. In particular, a similar methodology has been followed in [7] for building proof trees in natural deduction theorem provers written in Lambda Prolog.

Compared to the approach taken in GKPS, our combined use of *GENERIC* and *AUGMENT* search operations allows a rather more natural semantic analysis of unbounded dependencies. Indeed, under the GKPS approach, sentences containing gaps are treated differently from other sentences, since they are viewed from the beginning as λ-expressions whose λ-bound parameter has to be percolated itself down the tree, until it is consumed by the identity function introduced by the empty transition corresponding to the gap. By contrast, under our approach, sentences containing gaps are interpreted as normal sentences, with the only difference that their semantic interpretation is characterized by the occurrence of a generic in correspondence of the gap; the λ-expression on which a given filler (like a relative pronoun) must operate is then simply obtained by unifying the semantic interpretation of the sentence with a functional application of the form $(rel^{\sigma}\ c)$, where c is the generic and rel^{σ} is the desired λ-expression. Our approach is in this respect closer to the earlier GPSG development presented in [9], where sentences containing gaps are also treated in the same way as other sentences, and gaps within sentences correspond to occurrences of *designated variables* in the semantic interpretation. We could in fact say that our use of generics provides at last a rigorous formal characterization of this notion of designated variable. One of the reasons for which such a notion was abandoned in favour of the GKPS approach was indeed the lack of a clear understanding of its formal status [23].

3.3.3 Digression: Unused Hypotheses and Vacuous Abstraction

As mention in Chapter 2, the proof theory for *hhl* assumes implicitly the structural rule of Thinning, which allows for the possibility of not using all the hypotheses in a given program. As a consequence, any hypothesis which is introduced in terms of the *AUGMENT* search operation does not need to be consumed in the course of the remaining part of the proof; from the point of view of our proof-theoretic reconstruction of GPSG filler-gap dependencies, this means that a gap may be introduced "vacuously", thus leading to the possibility of accepting an ungrammatical string such as

*whom Paul married Kay

as is illustrated by the proof tree contained in figure 3.11. Now, notice that in figure 3.11 the semantic representation associated with the parsed string corresponds to a *vacuous* λ-expression, that is, an expression whose body does not contain any occurrence of the λ-parameter. In fact, the only possible parse for the substring

Paul married Kay

produces the semantic representation

$$\mathbf{((marry'\ Kay')\ Paul')}$$

and the only binding for the semantic representation rel^{σ} associated with relative clause is, in terms of the unification problem

$$(rel^{\sigma}\ \ c) = \mathbf{((marry'\ Kay')\ Paul')}$$

the vacuous λ-expression

$$\lambda np(\mathbf{(marry'\ Kay')\ Paul'})$$

Notice that, as distinct from the case of vacuous abstraction considered in the section above, this λ-expression cannot be ruled out via the *eigenvariable* condition, since there is no generic occurring in it.

$$\dfrac{\dfrac{\dfrac{\dfrac{\mathcal{P}_2 \cup \{(NP\ c)\} \Rightarrow (PN\ \textbf{Paul}')}{\mathcal{P}_2 \cup \{(NP\ c)\} \Rightarrow (NP\ \textbf{Paul}')}\,\textbf{(II)} \qquad \dfrac{\dfrac{\mathcal{P}_2 \cup \{(NP\ c)\} \Rightarrow (TV\ \textbf{married}') \qquad \mathcal{P}_2 \cup \{(NP\ c)\} \Rightarrow (NP\ \textbf{Kay}')}{\mathcal{P}_2 \cup \{(NP\ c)\} \Rightarrow (TV\ \textbf{married}') \wedge (NP\ \textbf{Kay}')}\,\textbf{(III)}}{\mathcal{P}_2 \cup \{(NP\ c)\} \Rightarrow (VP\ (\textbf{married}'\ \textbf{Kay}'))}\,\textbf{(II)}}{\mathcal{P}_2 \cup \{(NP\ c)\} \Rightarrow (NP\ \textbf{Paul}') \wedge (VP\ (\textbf{married}'\ \textbf{Kay}'))}\,\textbf{(III)}}{\mathcal{P}_2 \cup \{(NP\ c)\} \Rightarrow (S\ ((\textbf{married}'\ \textbf{Kay}')\ \textbf{Paul}'))}\,\textbf{(II)}$$

$$\dfrac{\mathcal{P}_2 \Rightarrow (NP\ c) \supset (S\ [(\lambda np((\textbf{married}\ \textbf{Kay}')\ \textbf{Paul}')\ c) \rightarrow_\beta ((\textbf{married}\ \textbf{Kay}')\ \textbf{Paul}')])}{\mathcal{P}_2 \Rightarrow \forall np[(NP\ np) \supset (S\ (\lambda np((\textbf{married}'\ \textbf{Kay}')\ \textbf{Paul}')\ np))]}\,\textbf{(VII)}\ \textbf{(VI)}$$

$$\dfrac{\mathcal{P}_2 \Rightarrow \forall np[(NP\ np) \supset (S\ (\lambda np((\textbf{married}'\ \textbf{Kay}')\ \textbf{Paul}')\ np))]}{\mathcal{P}_2 \Rightarrow (REL\ \lambda np((\textbf{married}'\ \textbf{Kay}')\ \textbf{Paul}'))}\,\textbf{(II)}$$

Figure 3.11: Proof tree for **whom Paul married Kay**

The solution must then perforce be metalogical, and can be worked out as follows. We have seen that our approach strictly follows the idea that derivations in sequent systems can be encoded as proof-terms, according to the formulae-as-types isomorphism. Indeed, steps of hypothesis introduction, obtained through the *AUGMENT* search operation, are encoded as corresponding λ-abstractions; cases of hypotheses introduced but never eliminated result in instances of vacuous abstraction, as in the example above. This gives us a well-defined way of filtering cases of vacuously introduced gaps, by modifying the rule for relative clauses in the previous section as follows:

$$\begin{aligned}
&\forall np[(NP\ z\ z\ np) \supset (S\ x\ y\ (rel\ np))]\ \wedge \\
&\neg(VACUOUS\ rel)\ \supset \\
&(REL\ (CONS\ \mathbf{whom}\ x)\ y\ rel)
\end{aligned}$$

The test for vacuousness can be straightforwardly implemented by exploiting certain properties of λ-terms, as will be shown in Chapter 5. Notice that the negation $\neg$ in the rule above corresponds to the metalogical principles of *negation as finite failure*, available in most Prolog implementations.

It remains the fact that this metalogical way of filtering vacuous gap introductions adds an element of aesthetical ugliness in the statement of the rules, and also an element of inefficiency in the computation, since it implies that there are parts of the proof which need to be fully generated and then discarded. Therefore, it would be of course quite more desirable to have a global condition of non-vacuousness in the gap-introducing rules. This could be achieved by postulating that the premise encoding the position of the gap, added to the program via the *AUGMENT* operation, *must* be used in the remaining part of the proof; or, in other terms, by postulating that it is not possible to *thin* over such a premise. Now, there is a logic, recently developed by the logician J. Y. Girard under the name of Linear Logic [13] [2], which specifically allows the possibility

[2] Linear Logic is itself related to Relevance Logic [2], a logic concerned with the philosophical problem of *relevant implication*. See [36] for an interesting discussion of the relationship between relevant implication and current linguistic theories. Another direct relation of Linear Logic is Lambek Calculus, on which we shall focus in Chapter 5.

of distinguishing between premises subject to the structural rules of Thinning and Contraction and premises which are not subject to such structural rules. It might be that in the future a similar capability of control over the structural rules will be incorporated in the *hhl* interpreter, thus providing a global way of ruling out vacuous gap introductions [3].

3.3.4 Rules Covering Filler-gap Dependencies - II

We now turn to the task of accounting for rules whose role is of specifying where a gap can go. As we pointed out earlier, such rules are here simply needed to constrain the distribution of gaps, as the complementary task of locating them on the frontier of a proof tree is already taken care of by the proof theory behind *hhl*. Thus, we want to be able to account for the fact that a sentence such as

*Fred loves a woman whom [Kay believes that $\uparrow$ married Paul]

is ungrammatical; given our rule for relative clauses in the sections above, such a sentence would instead be accepted as grammatical, since there is no constraint on where the gap can go inside a given sentence.

Under the usual GPSG approach, there are specific rules, obtained via application of metarules to the initial rules of the grammar, which interact with principles of feature percolation, taking care of the final destination of the gap. Thus, in the GKPS formalization of GPSG no application of metarules produces a rule licensing a gap in the subject position above, and therefore the sentence is ruled out as ungrammatical.

We obtain the same effect in our framework simply by replacing the introduction of gaps with the introduction of gap-terminating rules. Rules of this

[3]The introduction of such a control facility may have other important applications beside the linguistic one we are focusing on here. For instance, implementations of message-passing in the object-oriented style of programming can currently be obtained in *hhl* via a combined use of the *AUGMENT* and *GENERIC* operations, at the cost however of making use of metalogical operators such as Prolog's cut in order to forbid the access to hypotheses corresponding to a former state of a given object [30]. A more logical way of handling such hypotheses could be obtained by making them not subject to Contraction, thus implicitly forbidding their reuse.

kind define admissible constituent structures where a gap can occur, and at the same time they take care of the introduction of the generic which identifies the gap at the level of the semantic representation. The admissible constituent structures specified by such gap-terminating rules are going to be characterized by a missing daughter in correspondence of the gap - in other words, rather then explicitly introducing a gap, as we have been doing so far, they will introduce a "gapped" constituent structure. In this way, the only possible sites where a gap can occur will be limited to those allowed by the gap-terminating rules. For instance, the rule for relative clauses of the previous section can be constrained to allow gaps just from object positions by modifying it in the following way:

$$\begin{aligned}
&\forall np[(TV\ v\ z\ tv) \supset (VP\ v\ z\ (tv\ np)) \supset \\
&\quad (S\ x\ y\ (rel\ (tv\ np)))] \wedge \\
&\neg(VACUOUS\ rel) \supset \\
&(REL\ (CONS\ \textbf{whom}\ x)\ y\ \lambda np(rel\ (tv\ np)))
\end{aligned}$$

In this way, we embody in the gap-introducing rule itself a definite clause analogous of the GPSG gap-terminating rule

$$VP/NP \rightarrow TV\ \ NP/NP \qquad \lambda np TV'(NP/NP'(np))$$

in the form of the internal definite clause

$$(TV\ v\ z\ tv) \supset (VP\ v\ z\ (tv\ np))$$

which licences a gap NP after a transitive verb. This definite clause differs from the ordinary rule for transitive verbs

$$(TV\ v\ z\ tv) \wedge (NP\ z\ u\ np) \supset (VP\ v\ u\ (tv\ np))$$

in that there is no NP sister for the transitive verb. Moreover, the variable np corresponding to the semantic representation of the gap is bound by a universal quantifier having as scope the whole goal in which the added definite clause occurs, with the effect that a generic will be introduced to instantiate it.

Finally, observe that the variables corresponding to string positions in the gap-terminating definite clause are bound by (implicit) universal quantifiers having as scope the larger definite clause in the body of which the gap-terminating definite clause initially occurs; therefore, when such a definite clause is added to the program, its string positions arguments will be instantiated by logical variables, so that it will be possible to use the rule just to locate a single constituent structure. Thus, the situation does not change from what we have already in the gap-introducing rules of the previous sections.

With the same method, we can add to the grammar other rules for filler-gap dependencies with embedded internal rules accounting for legal gap sites.

Goal-orientedness and Modularity

From a computational point of view, embedding gap-terminating rules inside gap-introducing ones has the effect of making parsing more goal-oriented than in the standard GPSG formalization, where gap-terminating rules are independent from the gap-introducing ones and have all to be considered during a given computation, regardless of the fact that gaps have or have not been introduced. Instead, under our approach, a gap-terminating rule is temporarily added once a corresponding gap has been introduced and is afterwards removed once the same gap has been eliminated. In other words, while the metarule mechanism adds the gap-terminating rules globally and permanently, our system adds gap-terminating rules locally just with respect to a given proof; therefore, no gap-terminating rule needs ever to be considered when no filler-gap dependency is involved. An increased level of goal-orientedness in parsing follows directly from this situation.

Another way of looking at our handling of gap-eliminating rules is in terms of an increasingly modular organization of the grammar, directly exploitable at processing time in terms of the logical analysis of modules provided by Hereditary Harrop Logic [29]. Thus, we can view the gap-terminating rules locally introduced via gap-introducing ones as logic programs which are used by the logic program corresponding to the grammar when filler-gaps dependencies are

involved [4]. Clearly, in real life examples, such logic programs may well be of larger size then those considered here, where we are limiting ourselves to the case of a single gap-terminating rule; rather, we will have finite conjunctions of rules, corresponding to the set of possible sites where the introduced gap can be located. Thus, a given gap-introducing rule will operate by accessing a module $\mathcal{M}$, encodable as a finite conjunction of gap-terminating rules

$$Rule_1 \wedge \ \ldots \ \wedge Rule_n \quad , \quad 1 \leq n$$

$\mathcal{M}$ will be however unaccessible to rules which are not covering filler-gap dependencies.

3.3.5 Pied-Piping

Pied-Piping is given by the situation where the filler properly contains the relative pronoun. An example is given by the following sentence:

Fred, [the sister of whom] [Paul married $\uparrow$], loves Kay

The need for graph-theoretical percolation principles in order to account for unbounded depencencies has been brought forward by cases like these, the idea being that the filler noun-phrase inherits its status of *wh*-ness from the fact that the relative pronoun lies inside it. The GPSG analysis of the feature percolation is shown in the tree in Figure 3.12, with the $[+R]$ indicating the "relative" feature introduced by the pronoun. Our analysis differs here from the GKPS treatment of GPSG in that we do not rely on principles of feature percolation. Rather, we exploit the fact that we can have a separate rule for a possibly non-lexical filler. Such a rule, via a step of hypothesis introduction, also builds a λ-expression as semantic representation for the filler. In the rule for the relative clause, we then obtain the desired λ-expression by composing the λ-expression obtained by abstracting over the logical form associated with the sentence containing the gap and the λ-expression associated with the filler.

[4] A very interesting possibility to explore from this point of view would be revisiting GPSG metarules as tools for the automatic sinthesis of modules of this kind.

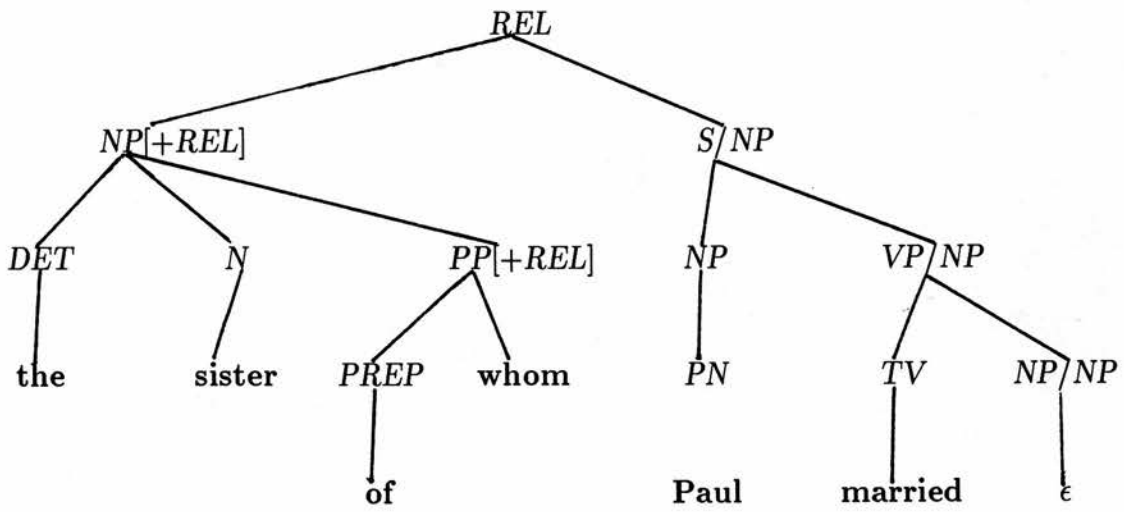

Figure 3.12: GPSG analysis for **the sister of whom Paul married**

$$\dfrac{\dfrac{\dfrac{\mathcal{P}_3 \cup (NP\ c) \Rightarrow (NP\ c)}{\mathcal{P}_3 \Rightarrow (NP\ c) \supset (NP\ [(\lambda np\, np\ c) \rightarrow_\beta c])}\ \textbf{(VII)}}{\mathcal{P}_3 \Rightarrow \forall np_1[(NP\ np_1) \supset (NP\ (\lambda np\, np\ np_1))]}\ \textbf{(VI)}}{\mathcal{P}_3 \Rightarrow (RELF\ \lambda np\, np)}\ \textbf{(II)}$$

Figure 3.13: Proof tree for **whom**

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\mathcal{P}_3 \cup \{(NP\ c)\} \Rightarrow (DET\ \textbf{the}')\quad \mathcal{P}_3 \cup \{(NP\ c)\} \Rightarrow (N\ \textbf{sister}') \wedge (PP\ (\textbf{of}'\ c))}{\mathcal{P}_3 \cup \{(NP\ c)\} \Rightarrow (DET\ \textbf{the}') \wedge ((N\ \textbf{sister}') \wedge (PP\ (\textbf{of}'\ c)))}\ \textbf{(III)}}{\mathcal{P}_3 \cup \{(NP\ c)\} \Rightarrow (NP\ ((\textbf{of}'\ c)(\textbf{the}'\ \textbf{sister}')))}\ \textbf{(II)}}{\Rightarrow (NP\ c) \supset (NP\ [(\lambda np((\textbf{of}'\ np)(\textbf{the}'\ \textbf{sister}'))\ c) \rightarrow_\beta ((\textbf{of}'\ c)(\textbf{the}'\ \textbf{sister}'))])}\ \textbf{(VII)}}{\mathcal{P}_3 \Rightarrow \forall np_1[(NP\ np_1) \supset (NP\ (\lambda np((\textbf{of}'\ np)(\textbf{the}'\ \textbf{sister}'))\ np_1))]}\ \textbf{(VI)}}{\mathcal{P}_3 \Rightarrow (RELF\ \lambda np((\textbf{of}'\ np)(\textbf{the}'\ \textbf{sister}')))}\ \textbf{(II)}$$

$$\dfrac{\dfrac{\mathcal{P}_3 \cup \{(NP\ c)\} \Rightarrow (PREP\ \textbf{of}')\quad \mathcal{P}_3 \cup \{(NP\ c)\} \Rightarrow (NP\ c)}{\mathcal{P}_3 \cup \{(NP\ c)\} \Rightarrow (PREP\ \textbf{of}') \wedge (NP\ c)}\ \textbf{(III)}}{\mathcal{P}_3 \cup \{(NP\ c)\} \Rightarrow (N\ \textbf{sister}')\quad \mathcal{P}_3 \cup \{(NP\ c)\} \Rightarrow (PP\ (\textbf{of}'\ c))}\ \textbf{(II)}$$

Figure 3.14: Proof tree for **the sister of whom**

This kind of analysis is closely related to one proposed by [48] in the framework of Combinatory Categorial Grammar, and we shall provide a similar one in the context of our *hhl* version of Categorial Grammar in the Chapter 5. (Our inferential machinery is however completely different from the one adopted by Steedman and Szabolczi, who make use of the combinatory rules of *functional composition* and *type raising*.)

On the other hand, we maintain the idea behind GKPS that pied-piping should be viewed as a generalization of the other cases of unbounded dependencies; in fact, the rule for the relative clause we give below can be viewed as a generalization of the rule given in the section above, where the fact that we have a single lexical entry as relative pronoun is semantically accounted for by having as filling λ-expression the identity function.

We introduce here a new category label *RELF* which will cover both the case of lexical fillers (like the relative pronoun *whom*) and of non-lexical fillers (like the expression *the sister of whom*) for relative clauses. Thus, let $\mathcal{P}_3$ be the program obtained by adding to $\mathcal{P}_1$ the following two clauses:

$$\forall np[(NP\ (CONS\ \mathbf{whom}\ x)\ (CONS\ \mathbf{whom}\ x)\ np) \supset$$
$$(NP\ x_1\ (CONS\ \mathbf{whom}\ x)\ (np_1\ np))] \wedge$$
$$\neg(VACUOUS\ np_1) \supset$$
$$(RELF\ x_1\ (CONS\ \mathbf{whom}\ x)\ np_1)$$

$$(RELF\ x_1\ x\ np_1) \wedge$$
$$\forall np[(TV\ v\ z\ tv) \supset (VP\ v\ z\ (tv\ np)) \supset$$
$$(S\ x\ y\ (rel\ (tv\ np)))] \wedge$$
$$\neg(VACUOUS\ rel) \supset (REL\ x_1\ y\ \lambda np(rel\ (tv\ (np_1\ np))))$$

Consider than the two sentences

whom Kay believes that Paul married

and

the sister of whom Kay believes that Paul married

According to the *RELF* rule above, the subproof for the filler *whom* is as in Figure 3.13, and the subproof for the for the filler *the sister of whom* is as in Figure 3.14. (Explicit representation of string positions and tests for vacuousness are as usual omitted.) In the first case the unification which provides us with the semantic representation of the filler is

$$(np_1^{\sigma}\ c) = c$$

which binds np_1^{σ} to the identity function

$$\lambda np\, np$$

In the second case, where pied-piping is involved, the unification which provides us with the semantic representation of the filler is

$$(np_1^{\sigma}\ c) = ((\mathbf{of'}\ c)(\mathbf{the'}\ \mathbf{sister'}))$$

which binds np_1^{σ} to the function

$$\lambda np((\mathbf{of'}\ np)(\mathbf{the'}\ \mathbf{sister'}))$$

Therefore, for the first relative clause we can obtain the same representation we had in Section 3.3.2, that is,

$$\lambda np((\mathbf{believe}'((\mathbf{marry}'\ np)\ \mathbf{Paul}'))\ \mathbf{Kay}')$$

while for the other one we obtain the representation

$$\lambda np((\mathbf{believe}'((\mathbf{marry}'\ ((\mathbf{of}\ np)(\mathbf{the}'\ \mathbf{sister}')))\ \mathbf{Paul}'))\ \mathbf{Kay}')$$

3.3.6 Gaps in Subject Position and Parasitic Gaps

Throughout this section, so far we have been just considering cases of a single gap occurring in object position. We conclude by indicating how to handle cases of gaps in subject position and of multiple gaps.

Gaps in Subject Position

Gaps in subject position can in certain cases be simply explained by allowing the filler to combine directly with an adjacent verb-phrase as in

the man who [↑ married Kay]

However, there are cases where the verb-phrase is not adjacent, as in

the man whom [Fred believes ↑ married Kay]

Such cases are accounted for in GPSG by allowing, through metarule application, gap-eliminating rules where a verb which normally would take a sentence as complement, takes instead a verb-phrase. Thus, the relative clause above would be accounted for in terms of the following terminating rule:

$$VP/NP \rightarrow STV \quad VP \qquad STV'(VP')$$

Notice that, as opposed to gaps in object position, this rule does not involve a "slashed" noun-phrase.

Obviously, there is no difficulty in stating an analogous rule in *hhl*, as shown below. Such a rule can then be appropriately embedded in a gap-introducing one.

$$(STV\ x\ z\ stv) \wedge (VP\ z\ y\ vp) \supset (VP\ x\ y\ (stv\ vp))$$

Parasitic Gaps

GPSG admits rules which percolate multiple occurrences of gaps, like the following:

$$VP/NP \rightarrow TV\ \ NP/NP\ \ ADVP/NP$$
$$\lambda np(ADVP'(np))(TV'(NP/NP'(np)))$$

A rule of this kind is needed to account for situations where we have multiple occurrences of the same gap, as in

articles which [I filed $\uparrow$ without reading $\uparrow$]

where the gap occurs in the verb-phrase and also "parasitically" in the adverbial which modifies it.

Although we do not state a specific rule here, constructions of this kind can be correspondingly accounted for in *hhl* by allowing the simultaneous introduction of two gaps sharing the same generic. Via embedded rules for gap-elimination, one such gap will then be located inside a verb-phrase, while the other will be located within an adjacent modifying adverbial. If we then omit the test for non-vacuous abstraction in the adverbial, then we can account for the optionality of the parasitic gap, as in

articles which [I filed $\uparrow$ without reading your instructions]

On the other hand, by enforcing non-vacuous abstraction in the verb-phrase we can rule out sentences such as

*instructions which [I filed articles without reading $\uparrow$]

where the gap occurs only in the adverbial.

Chapter 4

An Implicational Version of Gap Threading

4.1 Gap Threading

In the tradition of logic grammars there is a very important antecedent in the treatment of filler-gap dependencies, given by the technique of *gap threading* first introduced by Fernando Pereira in the context of the modification of DCGs known as Extraposition Grammars [41]. Such a technique has then reappeared under many different forms; see Pereira and Shieber [42] for an excellent recent illustration. Rather than being incompatible with the implicational treatment of gaps we have proposed in the previous chapter in the context of a logical re-examination of GPSG, the gap threading technique can blend quite well with it, thus offering an alternative manner of encoding various kinds of constraints, and gaining a natural way to build logical forms. We first illustrate how this technique works in the context of standard (Prolog-based) DCGs, we then point out its selling points and its shortcomings, and finally we show how to improve on such shortcomings by adding to it the notion of gaps as hypothetical constituents which was developed, with a different framework in mind, in the previous chapter.

$(NP\ x\ z\ f_1\ f_0) \wedge (VP\ z\ y\ f_0\ f) \supset (S\ x\ y\ f_1\ f)$

$(TV\ x\ z) \wedge (NP\ z\ y\ f_1\ f) \supset (VP\ x\ y\ f_1\ f)$

$(STV\ x\ z) \wedge (S\_BAR\ z\ y\ f_1\ f) \supset (VP\ x\ y\ f_1\ f)$

$(S\ x\ y\ f_1\ f) \supset (S\_BAR\ (CONS\ \textbf{that}\ x)\ y\ f_1\ f)$

$(DET\ x\ z) \wedge (N\ z\ y) \supset (NP\ x\ y\ f\ f)$

$(DET\ x\ z) \wedge (N\ z\ y) \wedge (PP\ y\ v\ f_1\ f) \supset (NP\ x\ v\ f_1\ f)$

$(PN\ x\ y) \supset (NP\ x\ y\ f\ f)$

$(PREP\ x\ z) \wedge (NP\ z\ y\ f_1\ f) \supset (PP\ x\ y\ f_1\ f)$

$(DET\ (CONS\ \textbf{the}\ l)\ l)$

$(N\ (CONS\ \textbf{sister}\ l)\ l)$

$(N\ (CONS\ \textbf{woman}\ l)\ l)$

$(PN\ (CONS\ \textbf{Kay}\ l)\ l)$

$(PN\ (CONS\ \textbf{Fred}\ l)\ l)$

$(PN\ (CONS\ \textbf{Paul}\ l)\ l)$

$(TV\ (CONS\ \textbf{loves}\ l)\ l)$

$(TV\ (CONS\ \textbf{married}\ l)\ l)$

$(STV\ (CONS\ \textbf{believes}\ l)\ l)$

$(PREP\ (CONS\ \textbf{of}\ l)\ l)$

Figure 4.1: Adding filler lists to the DCG in Figure 3.2

4.2 Gap Threading in Standard DCGs

The idea behind gap threading is that the gap information associated with a given constituent corresponds to the list of gaps whose corresponding fillers are not covered by it. This can be conveniently expressed by adding two extra-arguments to the non-terminal predicates, using the difference-list encoding analogous to the one adopted for string positions; the list consequently defined is what is called the *filler* list of the given constituent. Thus, consider for instance the simple DCG in Figure 3.2. We can add filler lists to the non-terminals in this grammar by restating it as in Figure 4.1.

We can then have a rule for relative clauses like the one below, which intro-

duces a noun-phrase marker in the filler list of the sentence adjacent to the relative pronoun. Such a marker is obtained by combining the first-order function symbol *GAP* with the constant **np** to obtain the first-order term (*GAP* **np**).

$$(S\ x\ y\ (CONS\ (GAP\ \mathbf{np})\ f)\ f) \supset (REL\ (CONS\ \mathbf{whom}\ x)\ y\ f\ f)$$

Moreover, we must assume that we have lexical entries for empty noun-phrases, corresponding to possible gaps. A lexical entry like the one below does just that.

$$(NP\ x\ x\ (CONS\ (GAP\ \mathbf{np})\ f)\ f)$$

We can then parse the relative clause *whom Kay believes that Paul married* as in the proof tree in Figure 4.2, from where as usual string positions are omitted, and only filler arguments are displayed. $\mathcal{P}$ refers here to the grammar in Figure 4.1 plus the rule for relative clauses and the lexical entry for empty noun phrases.

4.2.1 Expressing Constraints on the Distribution of Gaps

The gap threading technique provides a very straightforward method to impose constraints over the distribution of gaps. Such constraints can in fact be achieved in one of the following ways:

(i) omitting from certain constituents the arguments encoding filler lists

(ii) making empty the filler list of a given consituent

(i) has been used for the DCG in figure 4.1 to encode lexical entries for non-empty lexical material. *(ii)* has been used in 4.1 to express the fact that, for instance, no gap can occur within a noun-phrase obtained by combining a noun with a determiner. This is indeed the constraint imposed through a rule such

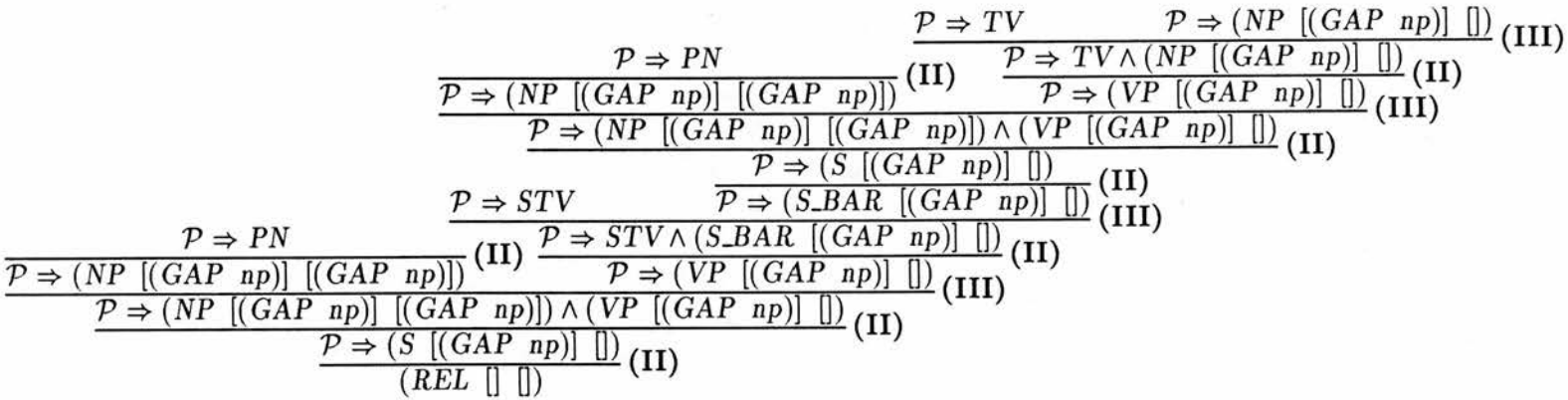

Figure 4.2: Proof tree for **whom Kay believes that Paul married**

as

$$(DET\ x\ z) \wedge (N\ z\ y) \supset (NP\ x\ y\ f\ f)$$

Thus, *(ii)* is in particular useful in expressing the fact that constituents of a given kind can be extracted from a certain context, but not from another. Indeed, suppose we want to further constrain the grammar in 4.1 so as to disallow gaps in subject position, or from within constituents in subject position. This can simply be achieved by modifying the rule for combining subject noun-phrases with verb-phrases into sentences as follows:

$$(NP\ x\ z\ f_1\ f_1) \wedge (VP\ z\ y\ f_1\ f) \supset (S\ x\ y\ f_1\ f)$$

As the subject noun-phrase now has an empty filler list, no gap can occur in that position.

4.2.2 The Problem of Semantic Interpretation

We come now to what seems to be the major problem of gap threading as implemented within standard DCGs, that is the fact that there is no obvious method to provide a natural semantic interpretation of filler-gap dependencies. In the end, such a problem has its roots in the lack in standard Prolog of a mechanism for expressing variable binding and λ-abstraction. Thus, there is no coherent way of obtaining a complex property encoded in the form of a λ-expression as the semantic interpretation of a constituent characterized by the occurrence of a gap.

Consider the solution which can be worked out in the semantic framework developed in [42], which we think is substantially in the right direction, but is implemented within a formal environment of inadequate expressive power. We can start by associating a universally quantified variable at the definite clause level as the semantic interpretation of the empty noun-phrase; such a variable will occur also inside the noun-phrase marker which goes into the filler list. (We

assume for this purpose that the function symbol *GAP* used to construct such a noun-phrase marker now takes also a second argument, corresponding to the semantic interpretation.)

$$(NP\ x\ x\ (CONS\ (GAP\ \mathbf{np}\ np_1)\ f)\ f\ np_1)$$

A similar semantic interpretation gets associated with the rule for relative clauses; observe that in this case the λ-operator in the logical form binds a variable which also occurs inside the filler list.

$$(S\ x\ y\ (CONS\ (GAP\ \mathbf{np}\ np_2)\ f)\ f\ s)\ \supset$$
$$(REL\ (CONS\ \mathbf{whom}\ x)\ y\ f\ f\ \lambda np_2\, s)$$

Now, in the course of a derivation, the filler list of the sentence will unify with the filler list of the noun-phrase, and so also will the two logical variables np_1^σ and np_2^σ to which np_1 and np_2 have been instantiated in the course of the proof; therefore, the body of the λ-expression will also be characterized by a corresponding occurrence of the unification of np_1^σ and np_2^σ. However, observe that this is obtained at the cost of contradicting the assumption that np_2 is really a λ-bound variable. Indeed, we must here rely on the fact that such a variable also occurs as bound by a universal quantifier having as scope the whole definite clause where the λ-expression itself occurs, and thus can get istantiated in the course of the proof. In the end, the trick works out just because we are dealing with a fake λ-operator, which in reality is but a first-order function symbol; the same trick does not work out in an environment based on genuine λ-terms, like λ-Prolog. So, in the rule above, if we make explicit the universal quantifier binding the np_2 and we use the usual notation for first-order function

symbols for λ, we end up with the following notation:

$$\forall np_2[(S\ x\ y\ (CONS\ (GAP\ \mathbf{np}\ np_2)\ f)\ f\ s) \supset$$
$$(REL\ (CONS\ \mathbf{whom}\ x)\ y\ f\ f\ (\lambda\ np_2\ \ s))]$$

So, under this approach, λ-terms do not have really a logical status of their own, but are rather treated metalogically as a particular kind of data. Therefore, the grammar writer has to take on her/himself the burden to check that things always go right during their manipulation. For instance, nothing forbids in principle that the pseudo λ-operator can be used to "bind" a non-variable expression; or, to mention a more subtle possibility, nothing forbids that a vacuous λ-abstraction becomes non-vacuous because the λ-bound variable gets unified with a variable in the body, as for instance would happen in solving the first-order unification problem

$$(\lambda\ x\ (F\ y)) = (\lambda\ x\ (F\ x))$$

Clearly, it would be much better to have a version of gap-threading where the grammar writer does not have to be concerned about this kind of problems, since they are already taken care of by the underlying logical mechanism governing the combination of λ-terms. In the rest of this chapter we show how this can be straightforwardly achieved with *hhl*.

4.3 Threading Hypothetical Constituents

Our *hhl*-based solution to the problems of semantic interpretation described above hinges on the possibility of threading hypothetical constituents introduced via rule **(VII)**, and of associating generics with such hypothetical constituents via proof rule **(VI)**. Of course, we shall also rely on the use of β-reduction.

4.3.1 Putting Generics in the Filler List

We start by showing how we can replace empty constituents in the lexicon with gaps introduced at run time; this will ultimately involve putting generics in

the filler lists of constituents. For the time being we do not worry about logical forms; we shall show later that an adequate treatment of semantic interpretation follows out immediately.

Thus, our first step is in modifying the rule for relative clauses of Section 4.2 in the following way:

$$\begin{aligned} &\forall np[(NP\ z\ z\ (CONS\ np\ f)\ f) \supset \\ &\quad (S\ x\ y\ (CONS\ np\ f)\ f)] \supset \\ &(REL\ (CONS\ \mathbf{whom}\ x)\ y\ f\ f) \end{aligned}$$

Clearly, this is very similar to the gap-introducing rules considered in the previous chapter, except that here there is no need of using gap-terminating rules, since constraints on the distribution of gaps can be taken care of as in Section 4.2.1. Also, notice that there is no need for a test to check that the introduced noun phrase gets actually used in the course of the proof, which in the previous chapter was obtained by checking against vacuous abstraction of λ-expressions corresponding to logical forms; in this case, the filler list of the sentence containing the gap must at some point unify with the filler list of the gap itself, with the obvious implication that the gap must be used in the derivation. An analogue of the proof tree in Figure 4.2 is now given by the proof tree in Figure 4.3., where the $\mathcal{P}_1$ refers to the grammar in Figure 4.1 plus the new rule for relative clauses, and c is the generic introduced via proof rule **(VI)**.

4.3.2 Pied-piping

We can further modify the rule above so that it takes care of pied-piped cases as well, in a similar way to the approach taken in the previous chapter. Here, however, we do not introduce a separate rule for the filler, but we collapse both the rule for the relative pronoun and that for the relative clause in a unique clause. Thus, we introduce two hypothetical noun phrases sharing the same filler list, which is also shared by the filler noun phrase left-adjacentent to the

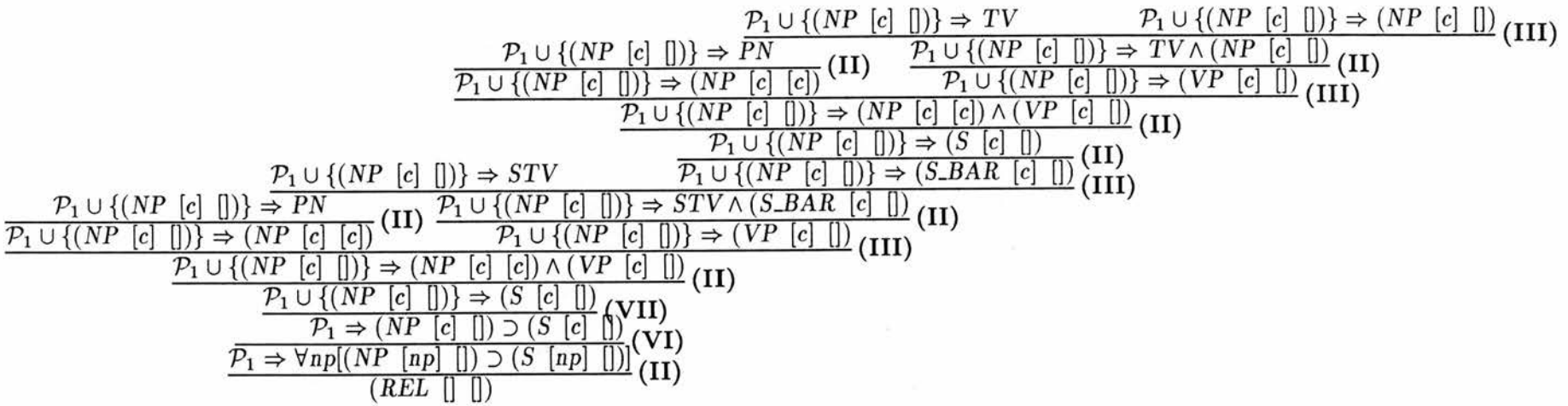

Figure 4.3: Proof tree for **whom Kay believes that Paul married**

relative pronoun and by the sentence right-adjacent to it. Figures 4.4-4.5 contain proofs involving a pied-pied and a non pied-piped example of relative clause. Here $\mathcal{P}_3$ refers to the grammar in 4.1 augmented with the rule below.

$$\begin{aligned}
&\forall np[((NP\ (CONS\ \textbf{whom}\ x)\ (CONS\ \textbf{whom}\ x)\ (CONS\ np\ f)\ f)\ \supset \\
&\quad (NP\ x_1\ (CONS\ \textbf{whom}\ x)\ (CONS\ np\ f)\ f))\ \wedge \\
&\quad ((NP\ z\ z\ (CONS\ np\ f)\ f)\ \supset \\
&\quad (S\ x\ y\ (CONS\ np\ f)\ f))]\ \supset \\
&(REL\ x_1\ y\ f\ f)
\end{aligned}$$

4.3.3 Semantic Interpretation

The problem of semantic interpretation can now be solved simply by making the generic introduced with the gap also its associated logical form; the logical form of the constituent where the gap occurs can then be analyzed as the application of a λ-expression to that same generic. In this way, we can carry over the spirit of the solution suggested in [42] within a correct logical setting. Thus, we can add logical forms arguments to the rule in the previous section as follows:

$$\begin{aligned}
&\forall np[((NP\ (CONS\ \textbf{whom}\ x)\ (CONS\ \textbf{whom}\ x)\ (CONS\ np\ f)\ f\ np)\ \supset \\
&\quad (NP\ x_1\ (CONS\ \textbf{whom}\ x)\ (CONS\ np\ f)\ f\ (np_1\ np)))\ \wedge \\
&\quad ((NP\ z\ z\ (CONS\ np\ f)\ f\ np)\ \supset \\
&\quad (S\ x\ y\ (CONS\ np\ f)\ f\ (rel\ np)))]\ \supset \\
&(REL\ x_1\ y\ f\ f\ \lambda np(rel\ (np_1\ np)))
\end{aligned}$$

The grammar in Figure 4.1 can also be modified in a similar way, as in Figure 4.6

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\mathcal{P}_2 \cup \{(NP\ [c]\ [])\} \Rightarrow TV \qquad \mathcal{P}_2 \cup \{(NP\ [c]\ [])\} \Rightarrow (NP\ [c]\ [])}{\mathcal{P}_2 \cup \{(NP\ [c]\ [])\} \Rightarrow PN}\ \textbf{(II)} \qquad \dfrac{\mathcal{P}_2 \cup \{(NP\ [c]\ [])\} \Rightarrow TV \wedge (NP\ [c]\ [])}{\mathcal{P}_2 \cup \{(NP\ [c]\ [])\} \Rightarrow (VP\ [c]\ [])}\ \textbf{(II)}}{\mathcal{P}_2 \cup \{(NP\ [c]\ [])\} \Rightarrow (NP\ [c]\ [c])}\ \textbf{(III)}}{\mathcal{P}_2 \cup \{(NP\ [c]\ [])\} \Rightarrow (NP\ [c]\ [c]) \wedge (VP\ [c]\ [])}\ \textbf{(II)}}{\mathcal{P}_2 \cup \{(NP\ [c]\ [])\} \Rightarrow (NP\ [c]\ [])}\ \textbf{(VII)} \qquad \dfrac{\mathcal{P}_2 \cup \{(NP\ [c]\ [])\} \Rightarrow (S\ [c]\ [])}{\mathcal{P}_2 \Rightarrow \{(NP\ [c]\ [])\} \supset (S\ [c]\ [])}\ \textbf{(VII)}}{\mathcal{P}_2 \Rightarrow (NP\ [c]\ []) \supset (NP\ [c]\ [])}\ \textbf{(III)}}{\mathcal{P}_2 \Rightarrow ((NP\ [c]\ []) \supset (NP\ [c]\ [])) \wedge ((NP\ [c]\ []) \supset (S\ [c]\ []))}\ \textbf{(VI)}}{\mathcal{P}_2 \Rightarrow \forall np[((NP\ [np]\ []) \supset (NP\ [np]\ [])) \wedge ((NP\ [np]\ []) \supset (S\ [np]\ []))]}\ \textbf{(II)}}{(REL\ []\ [])}$$

Figure 4.4: Proof tree for **whom Paul married**

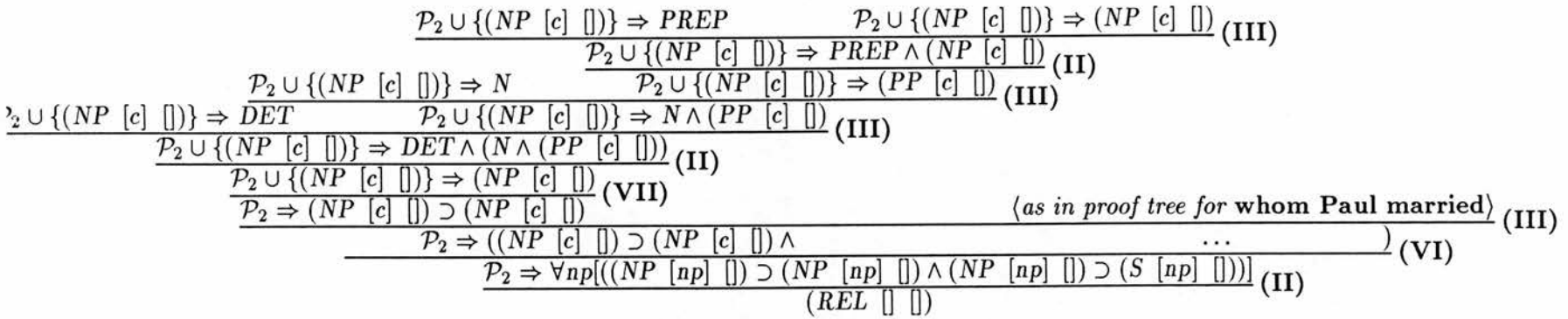

Figure 4.5: Proof tree for **the sister of whom Paul married**

$(NP\ x\ z\ f_1\ f_0\ np) \wedge (VP\ z\ y\ f_0\ f\ vp) \supset (S\ x\ y\ f_1\ f\ (vp\ np))$

$(TV\ x\ z\ tv) \wedge (NP\ z\ y\ f_1\ f\ np) \supset (VP\ x\ y\ f_1\ f\ (tv\ np))$

$(STV\ x\ z\ stv) \wedge (S\_BAR\ z\ y\ f_1\ f\ s\_bar) \supset (VP\ x\ y\ f_1\ f\ (stv\ s\_bar))$

$(S\ x\ y\ f_1\ f\ s) \supset (S\_BAR\ (CONS\ \textbf{that}\ x)\ y\ f_1\ f\ s)$

$(DET\ x\ z\ det) \wedge (N\ z\ y\ n) \supset (NP\ x\ y\ f\ f\ (\text{det}\ n))$

$(DET\ x\ z\ det) \wedge (N\ z\ y\ n)\ n) \wedge\ (PP\ y\ v\ f_1\ f\ pp) \supset$
$(NP\ x\ v\ f_1\ f\ (pp\ (\text{det}\ n)))$

$(PN\ x\ y\ pn) \supset (NP\ x\ y\ f\ f\ pn)$

$(PREP\ x\ z\ prep) \wedge (NP\ z\ y\ f_1\ f\ np) \supset (NP\ x\ y\ f_1\ f\ (prep\ np))$

$(DET\ (CONS\ \textbf{the}\ l)\ l\ \textbf{the}')$

$(N\ (CONS\ \textbf{sister}\ l)\ l\ \textbf{sister}')$

$(N\ (CONS\ \textbf{woman}\ l)\ l\ \textbf{womam}')$

$(PN\ (CONS\ \textbf{Kay}\ l)\ l\ \textbf{Kay}')$

$(PN\ (CONS\ \textbf{Fred}\ l)\ l\ \textbf{Fred}')$

$(PN\ (CONS\ \textbf{Paul}\ l)\ l\ \textbf{Paul}')$

$(TV\ (CONS\ \textbf{loves}\ l)\ l\ \textbf{love}')$

$(TV\ (CONS\ \textbf{married}\ l)\ l\ \textbf{married}')$

$(STV\ (CONS\ \textbf{believes}\ l)\ l\ \textbf{believe}')$

$(PREP\ (CONS\ \textbf{of}\ l)\ l\ \textbf{of}')$

Figure 4.6: Adding logical forms to the DCG in Figure 4.1

Chapter 5

Categorial Grammar and Parsing as Type Inference

5.1 Natural Language as a Typed Language

Categorial Grammar (CG) is a framework for natural language analysis of long-standing tradition, started during the thirties by a group of Polish logicians. Up until the seventies, CG enjoyed contributions of high theoretical interest from other logicians such as Bar-Hillel, Lambek and Montague, who were attracted by the formal elegance of this framework, but failed to attract the widespread attention of linguists. However, in the past few years contributions by linguists such as Steedman, Dowty, Moortgat, Bach, Oehrle and many others, have shown that CG can provide attractive accounts of interesting natural language constructions; in particular, CG provides a natural proof-theoretic account of unbounded dependencies phenomena strictly related to the feature-based one given in GPSG, without the complicated stipulations about feature percolation which go along with the GKPS formalization of GPSG [1]. Our intent in chapters 3 and 4 (as should become completely clear after reading the present one) was to show that the proof-theoretic flavour behind CG could indeed be imported in GPSG itself, and also in the gap-threading methodology coming from

[1] On the other hand, so far there has not been on the side of CG an effort comparable to that of GKPS in the systematic coverage of linguistic phenomena

the tradition of logic grammars, with the pleasant effect of creating an extended versions of DCG directly compatible with Hereditary Harrop Logic.

On the other hand, our treatment of CG here is going to be different in an important way from earlier ones. All the previous approaches were in fact based on the assumption that CG could provide a framework for the inference of syntactic types as in the propositional type systems for λ-Calculus and Combinatory Logic; however, in order to cope with the oddities of natural language, type systems in propositional CG are "specialized" in a fashion which takes them away from standard logic. The logics on which they are based (as theoretically interesting as they are) do not seem to be of use besides the application they were tailored for in terms of syntactic analysis [2]. By contrast, our approach will consist in accounting for the difficulties of natural language syntax not by specializing propositional logic, but rather by augmenting it with the power of quantification. As in the case of GPSG, *hhl* provides us with a suitable tool for this purpose. The main advantages we gain in this way are the following:

- we incorporate CG into a general purpose logic programming language, in the same way as phrase-structure grammars have been through the development of DCGs
- we use a logic characterized by a well defined semantics and a well understood proof theory
- we achieve a greater expressive power, which allows elegant and simple accounts for natural language constructions difficult to handle in propositional CG

[2] However, some of the restrictions of Lambek Calculus, the main predecessor and source of inspiration for our approach, could even usefully maintained in our system, if one could make without them when needed. Again, as we already did in chapter 2, we shall see that Linear Logic [13] offers an interesting direction of research from this point of view.

5.2 From Classical Categorial Grammar to Definite Clauses as Types

CG is directly inspired by the idea that natural language can be viewed as a *typed* functional language, as in the formal languages of typed λ-Calculus and Combinatory Logic, and in strongly typed programming languages. Thus, all linguistic information is encoded in the lexicon via the assignment of syntactic types to lexical items, and expressions are either functions or arguments; parsing can be naturally viewed as type inference, with the type of a grammatical expression inferrable from the types of its subexpressions.

However, as distinct from λ-Calculus and Combinatory Logic, where a function can only combine with an adjacent argument to its right, natural language is characterized by complex patterns of combination between functions and arguments, where arguments must combine to the right of certain functions and to the left of others, and where a function and its argument are not necessarily adjacent. We first consider how this problem has been addressed in the classical development of CG, and then we propose a novel notation, based on definite clause logic, which, while fully remaining in the spirit of the original formulation of CG, is both more expressive and more standard from a logical point of view. We then show how parsing can be implemented as type inference using Hereditary Harrop Logic.

5.2.1 Classical Categorial Grammar

In classical CG, syntactic types are viewed as expressions of an implicational calculus of propositions, where atomic propositions correspond to atomic types, and implicational propositions account for complex types. A string is grammatical if and only if its syntactic type can be logically derived from the types of its words, assuming certain inference rules.

Word-order constraints are obtained by having two symmetric forms of "directional" implication, usually indicated with the forward slash / and the backward slash \, constraining the antecedent of a complex type to be, respectively,

right- or left-adjacent. A word, or a string of words, associated with a right- (left-) oriented type can then be thought of as a right- (left-) oriented function looking for an adjacent argument of the type specified in the antecedent. A convention more or less generally followed by linguists working in this framework is to have the antecedent and the consequent of an implication respectively on the right and on the left of the connective. Thus, the type-assignment below says that the ditransitive verb **put** is a function taking a right-adjacent argument of type NP, to return a function taking a right-adjacent argument of type PP, to return a function taking a left-adjacent argument of type NP, to finally return an expression of the atomic type S.

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

Analogous type-assignments for, respectively, the noun phrase **Paul** and the transitive verb **married** are the following:

$$\textbf{Paul} \ : \ NP$$

$$\textbf{married} \ : \ (S\backslash NP)/NP$$

Higher-order Functions

Classical CG also contemplates the possibility of higher-order functions, which have been exploited in recent linguist work [1, 45, 46, 47, 48, 6, 37] to give attractive accounts of the grammar of extraction and coordination. Thus, a possible type-assignment for the relative pronoun **whom**, which makes it into a function from functions from NP to S to relative clauses[3], is the following:

$$\textbf{whom} \ : \ REL/(S/NP)$$

Clearly, in the type assignment above the type of the input function S/NP has the intuitive meaning of an internal implication, which was formally justified

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

in the late fifties by Lambek [25], who developed a sequent calculus for the bidirectional implicational logic on which classical CG is based. The basic insight behind Lambek's work is what has inspired our treatment of GPSG unbounded dependencies in the previous chapter, and will be maintained in our proof-theoretic treatment of CG in a standard (non-directional) logical setting. However, an important alternative with a similar ability to exploit higher-order functions is Steedman's Combinatory Grammar, which we shall briefly discuss later together with the propositional calculus developed by Lambek. Given such a similar ability, solutions to particular linguistic problems in terms of lexical type-assignments proposed by linguists working in one framework can be often transferred to the other framework; indeed, we shall give examples of definite clause versions of type-assignments proposed within Combinatory Grammar in our implementation of CG, which is proof-theoretically related to Lambek's propositional calculus.

5.2.2 Definite Clauses as Types

The DCG framework, which allows the possibility of encoding phrase-structure grammars as sets of definite clauses, and, if extended in terms of Hereditary Harrop Logic, of incorporating the GPSG notation for unbounded dependencies, suggests a more expressive alternative for encoding word-order constraints than the one followed in classical CG. Such an alternative eliminates all notions of directionality from the logical connectives, and any explicit requirement of adjacency between functions and arguments, and replaces propositions with quantified definite clauses.

Thus, atomic types are viewed as atomic formulae obtained from predicates taking as arguments string positions, plus other optional arguments corresponding to logical forms, agreement features etc.. We distinguish, as usual, two obligatory string arguments corresponding, respectively, to the left- and right-end of a given string, and we use the list notation [] to refer to the empty string. Therefore, if w is a string which is a grammatical sentence in the language obtained from a certain lexicon, then its (ground) type will be

$$(S\ w\ []\ \ldots\ )$$

This type is an atomic definite clause where the left-end string argument corresponds to w itself, and the right-end string argument correspond to the empty string. As we shall show, constraints over the order of constituents are enforced by sharing string variables across subformulae inside complex (functional) types.

5.3 Type Assignments with Definite Clauses

5.3.1 Exploiting Logical Variables in Program Clauses

The definite clauses whe shall use in this chapter for type assignments will exploit even more systematically than in the previous chapters the fact that in *hhl* it is possible to have occurrences of logical variables in program clauses.

In our proof-theoretical account of CG, all definite clauses used during a certain proof will be characterized by the fact that the arguments for string positions are filled with logical variables. Thus, we shall view definite clauses assigned as types to *occurrences of lexical items* in a string as corresponding to unique ground instantiations of their string variables; such ground instantiations will be made explicit via unification in the process of proving the grammaticality of a given input string. As we shall see further on, such instantiated definite clauses will be obtained before the parsing process starts, at "lexical retrieval" time, via instantiation with logical variables of the types assigned to the *lexical items* in the lexicon. Throughout this section, we shall assume that we are working with already instantiated definite clauses.

5.3.2 Type-assignments to Constants and First-order Functions

The following are possible type-assignments for (*occurrences of*) the noun-phrase **Paul**, the transitive verb **married** and the ditransitive verb **put**:

$$\textbf{Paul} \quad : \quad (NP\ (CONS\ \textbf{Paul}\ x^{\sigma})\ x^{\sigma}\ \textbf{Paul}')$$

$$\textbf{married} \quad : \quad \forall np_2[(NP\ x^{\sigma}\ (CONS\ \textbf{married}\ z^{\sigma})\ np_1^{\sigma}) \wedge (NP\ z^{\sigma}\ y^{\sigma}\ np_2) \supset (S\ x^{\sigma}\ y^{\sigma}\ ((\textbf{married}'\ np_2)\ np_1^{\sigma}))]$$

$$\begin{aligned}\textbf{put} \quad : \quad \forall np_2 \forall pp[&(NP\ x^{\sigma}\ (CONS\ \textbf{put}\ z^{\sigma})\ np_1^{\sigma}) \wedge \\ &(NP\ z^{\sigma}\ (CONS\ \textbf{on}\ v^{\sigma})\ np_2) \wedge \\ &(PP\ (CONS\ \textbf{on}\ v^{\sigma})\ y^{\sigma}\ pp) \supset \\ &(S\ x^{\sigma}\ y^{\sigma}\ ((\textbf{put}'\ ((pp\ np_2))\ np_1^{\sigma}))]\end{aligned}$$

All of such assignments convey the same syntactic information concerning word-order of constituents as the corresponding ones in the notation of classical CG; however, such information is obtained via sharing of string positions rather than in terms of directional connectives.

Semantic Arguments and Constraints over Gaps

An important point here is the treatment of the semantic arguments in the verbs. We can see in fact the variable for the logical form in the subject noun-phrase of the verbs **married** and **put** is itself a logical variable. Clearly, no generic parameter introduced via the *GENERIC* operation in the process of proving a universally quantified goal will be able to unify with this variable; such a unification would in fact imply a violation of the *eigenvariable* condition. Therefore, since we shall associate generics (as in the previous chapters) with gaps, no gap will be capable of "entering" the subject position of these verbs. We have in this way another simple and powerful method to constrain the distribution of gaps.

Flat versus Nested Notation

Notice also that the type assignments above assume the usual definite clause syntax where the body of a clause is "flattened" in a conjunction of formulae; classical CG assumes instead a "nested implications" notation. However, observe that a definite clause of the form

$$P_1 \wedge \ \ldots \ \wedge P_n \supset Q$$

is logically equivalent to an expression of the form

$$(P_1 \supset \ \ldots \ (P_n \supset Q) \ \ldots \ )$$

Thus, the definite clause notation used for our type-assignment is itself logically equivalent to a "nested implication" one which directly mirrors the original notation for type assignments in classical CG.

5.3.3 Higher-order Functions and Internal Implications

We can immediately translate the propositional type-assignment in section 5.2.1 into the following definite clause type-assignment, also characterized by an internal implication:

$$\begin{aligned} \textbf{which} \ : \ & \forall np[(NP \ y^\sigma \ \ y^\sigma \ \ np) \supset ((S \ x^\sigma \ \ y^\sigma \ \ (rel^\sigma \ \ np)) \wedge \\ & \neg(VACUOUS \ rel^\sigma))] \ \supset \\ & (REL \ \ (CONS \ \textbf{which} \ x^\sigma) \ \ y^\sigma \ \ rel^\sigma) \end{aligned}$$

Such a type-assignment requires the gap to occur at the end of the relative clause, as classical CG also does. Moreover, the internal universal quantifier provides a "generic" logical form for the gap, and there is an analogous test for the non-vacuousness of the λ-expression associated with the relative clause.

On the other hand, we can make the type above more general, so as to achieve greater expressive power with respect to classical CG. We give two examples of how this pays off in analyzing certain cases of extraction.

Non-peripheral Extraction

Both the propositional type for **which** from classical CG and the definite clause one above are adequate to describe the kind of constituent needed by a relative pronoun in the following right-oriented case of *peripheral* extraction, where the extraction site is located at one end of the sentence. (We indicate the extraction site with an upward-looking arrow.)

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

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

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

is difficult to describe in bidirectional propositional CG, where all functions must take left- or right-adjacent arguments. For instance, a solution like the one proposed in [46] involves permuting the arguments of a given function. Such an operation needs to be rather cumbersomely constrained, lest it should wildly overgenerate. Another solution, proposed in [37], is also cumbersome and counterintuitive, in that involves the assignment of multiple types to *wh*-expressions, one for each site where extraction can take place.

On the other hand, the greater expressive power deriving from predicate logic allows us to simply generalize our definite clause type-assignment in such a way that the variable identifying the extraction site ranges over the string positions in between the left- and right-end of the sentence on which the relative pronoun operates. This is obtained simply by relaxing the requirement that the position of the gap coincides with the end of the sentence. Thus, we have the following type-assignment:

$$\begin{aligned}\textbf{which} \;:\; & \forall np[(NP\ z^\sigma\ z^\sigma\ np) \supset ((S\ x^\sigma\ y^\sigma\ (rel^\sigma\ np)) \wedge \\ & \neg(VACUOUS\ rel^\sigma))] \\ \supset\ & (REL\ (CONS\ \textbf{which}\ x^\sigma)\ y^\sigma\ rel^\sigma)\end{aligned}$$

Pied-piping

We can generalize further the type-assignments for *wh*-expressions to make them capable of accounting for pied-piping cases; again, there is an immediate similarity with the GPSG-style rules of the previous section. In both cases, the inspiration is the same, that is, the possibility in propositional CG of providing an alternative assignment to relative pronouns in order to account for situations where the relative pronoun occurs within the filler, as in

the sister of whom [Paul married $\uparrow$]

A solution of this kind is proposed by Steedman and Szabolczi [48] in the framework of Combinatory Categorial Grammar, where, beside the propositional type-assignment in section 5.2.1, the relative pronoun is also viewed as a function from right-oriented functions from noun phrases to noun phrases to functions from "gapped" sentences to relative clauses, as in the following assignment:

$$\textbf{whom} \ : \ ((REL/(S/NP))\backslash(NP/NP))$$

We can transfer this kind of treatment of pied-piping to our framework as a simple generalization of the type-assignment in the previous section; thus, the assignment below will cover pied-piped and not pied-piped cases.

$$\begin{aligned}\textbf{whom} \ : \ \forall np[(((NP\ v^\sigma\ v^\sigma\ np)\ \supset\ & \\ ((NP\ x^\sigma\ (CONS\ \textbf{whom}\ w^\sigma)\ (np_1\ np))\ \wedge\ & \\ \neg(VACUOUS\ np_1^\sigma)))\ \wedge\ & \\ ((NP\ z^\sigma\ z^\sigma\ np)\ \supset\ & \\ ((S\ w^\sigma\ y^\sigma\ (rel^\sigma\ np))\wedge\neg(VACUOUS\ rel^\sigma))))]\ \supset\ & \\ (REL\ x^\sigma\ y^\sigma\ \lambda np_2(rel^\sigma\ (np_1^\sigma\ np_2)))\end{aligned}$$

This type assignment is clearly related to the solutions for pied-piping we had in the previous chapters. Indeed, as in the solutions for GPSG and gap-threading style DCGs, the combined power of quantification and β-reduction allows for

the case where the noun-phrase modifier is the empty string, and the identity function is its associated logical form.

Subject Extraction

We consider finally how to account for cases of subject extraction, which, although generally disallowed, are permitted in certain syntactical contexts. Cases where the filler combines directly with an adjacent verb-phrase as in

who [↑ married Kay]

are straightforward to handle, in that they simply involve a type assignment to *wh*-expressions which licenses such a combination. That is, all what is needed is to make a filler a function taking functions from subject noun-phrases to sentences and returning relative clauses.

More problematic are cases where the subject gap is internal. Thus, consider the following two sentences:

*who [Fred believes that ↑ married Kay]

who [Fred believes ↑ married Kay]

The first example is correctly ruled out as ungrammatical by assuming that the logical forms of subjects in the types assigned to occurrences of verbs correspond to logical variables, and that the verb **believe** is assigned the following type:

$$\begin{aligned}\textbf{believe} \;:\; \forall s[&(NP\ x^{\sigma}\ (CONS\ \textbf{believe}\ (CONS\ \textbf{that}\ z^{\sigma}))\ np^{\sigma}) \wedge \\ &(S\ z^{\sigma}\ y^{\sigma}\ s) \supset \\ &(S\ x^{\sigma}\ y^{\sigma}\ ((\textbf{believe}'\ s)\ np^{\sigma}))]\end{aligned}$$

Thus, since no generic can enter the subject of a sentence, subject extraction from that-complements is automatically forbidden.

We could have a similar type-assignment to account for the situation in which **believe** combines with a sentence not preceded by the complementizer **that**; however, in this way we would incorrectly rule out the possibility of grammatical cases of extraction like the second example above. A solution proposed by Steedman [46] to cope with a similar situation in Combinatory Grammar consists in having for **believe**, besides the assignment involving as argument a sentence preceded by the complementizer, a non-complementizer type-assignment where, in place of a sentence, the verb takes as arguments a subject noun-phrase followed by a verb-phrase. Thus, we have the following propositional type [4]:

$$\textbf{believe} \ : \ ((S\backslash NP)/(S\backslash NP))/NP$$

We can directly translate this type-assignment in our framework as follows:

$$\begin{aligned}\textbf{believe} \ : \ \forall np_1 \forall vp[&(NP \ x^\sigma \ (CONS \ \textbf{believe} \ z^\sigma) \ np^\sigma) \ \wedge \\ &(NP \ z^\sigma \ w^\sigma \ np_1) \ \wedge \\ &((NP \ w^\sigma \ w^\sigma \ np_2^\sigma) \supset (S \ w^\sigma \ y^\sigma \ (vp^\sigma \ np_2^\sigma)) \ \wedge \\ &\neg(VACUOUS \ vp)) \ \supset \\ &(S \ x^\sigma \ y^\sigma \ ((\textbf{believe}' \ (vp \ np_1)) \ np^\sigma))]\end{aligned}$$

Being universally quantified, the variable for the semantic representation in the internal subject noun-phrase can get instantiated by a generic, thus allowing for a gap in that position; nonetheless, we obtain the same logical form for the internal sentence we would have obtained had we parsed it as usual in "one chunk only". Notice also that the antecedent in the internal implication does not introduce a generic in its logical form, as in the case of implications used for gap-introduction; would it be so, the consequent in the same implication must fail, as, again, no generic can enter its subject position, assuming that all types assigned to occurrences of verbs have logical variables for the semantic

[4] This solution is somehow in the spirit of the GPSG approach to the same kind of problem considered in Chapter 3. However, it might be that further restrictions are here needed, so as to prevent unwanted passivizations such as **Bill was believed married Kay.*

representation of their subject arguments. Finally, the test to filter vacuous abstractions is here in the first place needed to constrain metalogically unification with built-in β-reduction, since there is no implicit constraint deriving from the interaction with the *eigenvariable* condition, given the fact that the noun-phrase introduced with the *AUGMENT* operation is not associated with a generic.

5.4 Type Inference

We implement parsing as type inference in our version of CG simply by trying to prove that there exists a proof for a sequent whose left part is given by the instantiated types associated in the lexicon with the words in the string and whose right part is the type of the string itself.

Under this approach, parsing consists of two steps, one corresponding to the definition of a logic program, and the other in the use of the same program for type inferencing[5]. The first steps involves instantiating with logical variables the λ-terms associated with the lexical items in the input string. For this purpose, we introduce the following auxiliary notation. Let l be a λ-term. Then

- $\Sigma(l) = \{l' \mid l' \in \bigcup_{\Sigma([x/t]\, l'')}$ if $l = \lambda x\, l''\}$
- $\Sigma(l) = \{l\}$ otherwise

Thus, $\Sigma(l)$ corresponds to the set of term-instantiations of l with respect to variables bound by outermost occurrences of the λ-operator. As usual, we can view procedurally the instantiated variables as coming into existence as logical variables, which then get specified into actual terms via unification.

We can now state the two-step parsing process as follows:

(i) Let α be a string $a_1 \ldots a_n$ of words from a lexicon $\mathcal{L}$. Then α defines a program $\mathcal{P}_\alpha = D'_1, \quad \ldots \quad , D'_n$ if and only if for each i, $1 \leq i \leq n$, $a_i : D_i \in \mathcal{L}$ and $D'_i \in \Sigma(D_i)$

[5]A two-step approach to parsing which may have an interesting relationship with the one proposed here has been recently developed for the TAG framework in [21].

(ii) Let $\mathcal{P}_\alpha$ be a program defined by a string α. Then α has type G_α if the sequent $\mathcal{P}_\alpha \Rightarrow G_\alpha$ has a proof

Observe that in case there is more than one occurrence of the same word in the input string, then several instantiated occurrences of the same type can also occur more than once in the program; all such different occurrences will indeed be characterized by distinct logical variables.

5.4.1 Some Examples

We give here some examples of type inferences using the approach above. Beside illustrating them in terms of proof trees, as in the previous chapters, we show also their effect on the programs they make use of, in terms of the unifications of logical variables in definite clauses. As in the previous chapter, we omit from the proof trees an explicit representation of the string positions, and of the metalogical tests for the non-vacuousness of λ-terms.

A Declarative Sentence

Consider the simple declarative sentence

A man married a woman

Let us assume we have the following type-assignments in the lexicon:

$$\mathbf{a} \quad : \quad \lambda x \lambda y \forall n[(N\ x\ y\ n) \supset (NP\ (CONS\ \mathbf{a}\ x)\ y\ (\mathbf{a}'\ n))]$$

$$\mathbf{man} \quad : \quad \lambda x[(N\ (CONS\ \mathbf{man}\ x)\ x\ \mathbf{man}')]$$

$$\mathbf{woman} \quad : \quad \lambda x[(N\ (CONS\ \mathbf{woman}\ x)\ x\ \mathbf{woman}')]$$

$$\mathbf{married} \quad : \quad \lambda x \lambda y \lambda z \lambda np_1 \forall np_2[(NP\ x\ (CONS\ \mathbf{married}\ z)\ np_1) \wedge (NP\ z\ y\ np_2) \supset (S\ x\ y\ ((\mathbf{married}'\ np_2)\ np_1))]$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\mathcal{D}_1 \Rightarrow (N\ \mathbf{man'})}{\mathcal{D}_1 \Rightarrow (NP\ (\mathbf{a'}\ \mathbf{man'}))}\,(\mathbf{II}) \qquad \dfrac{\mathcal{D}_1 \Rightarrow (N\ \mathbf{woman'})}{\mathcal{D}_1 \Rightarrow (NP\ (\mathbf{a'}\ \mathbf{woman'}))}\,(\mathbf{II})}{\mathcal{D}_1 \Rightarrow (NP\ (\mathbf{a'}\ \mathbf{man'})) \wedge (NP\ (\mathbf{a'woman'}))}\,(\mathbf{III})}{\mathcal{D}_1 \Rightarrow (S\ ((\mathbf{married'}\ (\mathbf{a'}\ \mathbf{woman'}))\ (\mathbf{a'}\ \mathbf{man'})))}\,(\mathbf{II})}{\Rightarrow \mathcal{D}_1 \supset (S\ ((\mathbf{married'}\ (\mathbf{a'}\ \mathbf{woman'}))\ (\mathbf{a'}\ \mathbf{man'})))}\,(\mathbf{VII})$$

Figure 5.1: Proof tree for **a man married a woman**

A program $\mathcal{P}_1$ determined by this sentence will consist of two instantiations of the type associated with the word **a**, and of an instantiations of each of the other types. A proof tree for the sequent

$$\mathcal{P}_1 \Rightarrow$$
$$(S\ [\mathbf{a\ man\ married\ a\ woman}]\ ((\mathbf{married'}\ (\mathbf{a'}\ \mathbf{woman'}))\ (\mathbf{a'}\ \mathbf{man'})))$$

is given in Figure 5.1. At the end of the parsing process the definite clauses in $\mathcal{P}_1$ have been fully instantiated as in Figure 5.2.

A Case of Pied-piping

Consider now the following relative clause, characterized by pied-piping:

$$\text{the sister of whom [Paul married } \uparrow \text{]}$$

Let us assume we have the following type-assignments in the lexicon:

$$\mathbf{the} \quad : \quad \lambda x \lambda y \forall n[(N\ x\ y\ n) \supset (NP\ (CONS\ \mathbf{the}\ x)\ y\ (\mathbf{the'}\ n))]$$

$$\mathbf{sister} \quad : \quad \lambda x[(N\ (CONS\ \mathbf{sister}\ x)\ x\ \mathbf{sister'})]$$

$$\mathbf{of} \quad : \quad \lambda x \lambda y \lambda z \lambda n \forall np[(N\ x\ (CONS\ \mathbf{of}\ z)\ n) \wedge (NP\ z\ y\ np) \supset (N\ x\ y\ ((\mathbf{of'}\ np)\ n))]$$

a : $\forall n[(N$ **[man married a woman]** **[married a woman]** $n) \supset$
$(NP$ **[a man married a woman]** **[married a woman]** (**a′** $n))]$

man : $(N$ **[man married a woman]** **[married a woman]** **man′**)

married : $\forall np_2[(NP$ **[a man married a woman]** **[married a woman]** (**a′** **man′**))
$(NP$ **[a woman]** [] $np_2) \supset$
$(S$ **[a man married a woman]** [] ((**married′** np_2) (**a′** **man′**)))]

a : $\forall n[(N$ **[woman]** [] $n) \supset (NP$ **[a woman]** [] (**a′** $n))]$

woman : $(N$ **[woman]** [] **woman′**)

Figure 5.2: Instantiation of the lexical types in **a man married a woman**

$$\textbf{whom} : \lambda x\lambda y\lambda z\lambda v\lambda w\lambda np_1\lambda rel\forall np[(((NP\ v\ v\ np) \supset ((NP\ x\ (CONS\ \textbf{whom}\ w)\ (np_1\ np)) \wedge \neg(VACUOUS\ np_1))) \wedge ((NP\ z\ z\ np) \supset ((S\ w\ y\ (rel\ np)) \wedge \neg(VACUOUS\ rel))))] \supset (REL\ x\ y\ \lambda np_2(rel\ (np_1\ np_2)))$$

$$\textbf{Paul} : \lambda x[(NP\ (CONS\ \textbf{Paul}\ x)\ x\ \textbf{Paul}')]$$

$$\textbf{married} : \lambda x\lambda y\lambda z\lambda np_1\forall np_2[(NP\ x\ (CONS\ \textbf{married}\ z)\ np_1) \wedge (NP\ z\ y\ np_2) \supset (S\ x\ y\ ((\textbf{married}'\ np_2)\ np_1))]$$

Parsing of the string above is obtained in terms of a proof for the sequent

$\mathcal{P}_2 \Rightarrow$
$(REL\ [\textbf{the sister}\ \ldots\ ]\ []\ \lambda np((\textbf{married}'\ (\textbf{the}'\ ((\textbf{of}'\ np)\ \textbf{sister}'))))\ \textbf{Paul}'))$

where $\mathcal{P}_2$ contains the instantiations of the types above as in Figure 5.3.

5.5 Comparison with Related Work

We now compare our version of CG with other related frameworks. We shall consider in particular Lambek's calculus of propositional bidirectional types, which offers a direct ground for comparison, but we shall also touch Combinatory Grammar and the recent augmentation of the classical CG notation known as Categorial Unification Grammar, which is also directly related to the framework presented here.

a : $\forall n[(N$ [**sister of whom** ...] [**of whom** ...] $n) \supset$
$(NP$ [**the sister of whom** ...] [**of whom** ...] $n))]$

sister : $(N$ [**sister of whom** ...] [**of whom** ...] **sister′**$)$

of : $\forall np[(N$ [**sister of whom** ...] [**of whom** ...] **sister′**$) \wedge$
$(NP$ [**whom** ...] [**whom** ...] $np) \supset$
$(N$ [**sister of whom** ...] [**whom** ...] $((\textbf{of}'\ np)$ **sister′**$))]$

whom : $\forall np[(((NP$ [**whom Paul married**] [**whom Paul married**] $np) \supset$
$((NP$ [**the** ...] [**whom** ...] **(the′** $((\textbf{of}'\ np)$ **sister′**$))) \wedge$
$\neg(VACUOUS\ \lambda np(\textbf{the}'\ ((\textbf{of}'\ np)))) \wedge$
$((NP$ [] [] $np) \supset$
$((S$ [**Paul married**] [] $((\textbf{married}'\ np)$ **Paul′**$)) \wedge$
$\neg(VACUOUS\ \lambda np((\textbf{married}'\ np)\ \textbf{Paul}'))))))] \supset$
$(REL$ [**the sister of whom Paul married**] []
$\lambda np((\textbf{married}'\ np)\ \textbf{Paul}'))$

Paul : $(NP$ [**Paul married** ...] [**married** ...] **Paul′**$)$

married : $\forall np_2[(NP$ [**Paul married**] [**married**] **(a′ man′**$)) \wedge$
$(NP$ [] [] $np_2) \supset$
$(S$ [**Paul married**] [] $((\textbf{married}'\ np_2)$ **(a′ Paul′**$)))]$

Figure 5.3: Instantiation of lexical types in **the sister of whom** ...

5.5.1 Lambek Calculus

Lambek's pioneering work [25] shows how the slash notation adopted in CG can be rigorously interpreted in proof-theoretic terms by developing a sequent calculus, known as Lambek Calculus, of bidirectional propositional types; he also introduces the connective $*$ for type concatenation.

Lambek Calculus has the following proof rules, where the sequence of hypotheses on the left of the sequent arrow is assumed to be always non-empty, in case this is not already made clear by the context:

for all types P, $P \Rightarrow P$ is an axiom

$$\frac{P_1, \ \ldots \ , P_i, X \Rightarrow Z \qquad P_{i+1}, \ \ldots \ , P_n \Rightarrow Y}{P_1, \ \ldots \ , P_i, X/Y, P_{i+1}, \ \ldots \ , P_n \Rightarrow Z} \ /\text{-left}$$

$$\frac{P_1, \ \ldots \ , P_n, Y \Rightarrow X}{P_1, \ \ldots \ , P_n \Rightarrow X/Y} \ /\text{-right}$$

$$\frac{P_1, \ \ldots \ , P_i \Rightarrow Y \qquad X, P_{i+1}, \ \ldots \ , P_n \Rightarrow Z}{P_1, \ \ldots \ , P_i, X\backslash Y, P_{i+1}, \ \ldots \ , P_n \Rightarrow Z} \ \backslash\text{-left}$$

$$\frac{Y, P_1, \ \ldots \ , P_n \Rightarrow X}{P_1, \ \ldots \ , P_n \Rightarrow X\backslash Y} \ \backslash\text{-right}$$

$$\frac{P_1, \ \ldots \ , X, Y, \ \ldots \ , P_n \Rightarrow Z}{P_1, \ \ldots \ , X * Y, \ \ldots \ , P_n \Rightarrow Z} \ *\text{-left}$$

$$\frac{P_1, \ \ldots \ , P_i \Rightarrow X \qquad P_{i+1}, \ \ldots \ , P_n \Rightarrow Y}{P_1, \ \ldots \ , P_i, P_{i+1}, \ \ldots \ , P_n \Rightarrow X * Y} \ *\text{-right}$$

Notice that, given the fact that bidirectional CG admits two implication connectives, one left-oriented and the other right-oriented, also each of the rules for implication come in two different flavors. Clearly, the /-left and \-left play operationally a similar role to the rule **(II)** in *hhl*, corresponding to the *BACKCHAIN* search operation, while the /-right and the \-right are similar to the rule **(VII)**, responsible for the *AUGMENT* search operation. Thus, the Lambek Calculus has a capability of accounting for unbounded dependencies in terms of implications on the right of the sequent arrow, which is the fundamental insight that we have transferred to our treatment of GPSG and CG. So, for instance, Lambek Calculus obtains the proof in figure 5.4 for the sequent

$$\dfrac{\dfrac{NP \Rightarrow NP \qquad S \Rightarrow S}{NP \;\; S\backslash NP \Rightarrow S}\backslash\text{-left} \qquad NP \Rightarrow NP}{\dfrac{NP \;\; (S\backslash NP)/NP \;\; NP \Rightarrow S}{NP \;\; (S\backslash NP)/NP \Rightarrow S/NP}/\text{-right}}/\text{-left}$$

Figure 5.4: Example of a proof in Lambek Calculus

$$NP \;\; (S\backslash NP)/NP \;\Rightarrow\; S/NP$$

where we use a notation for proof trees in Lambek Calculus similar to the one adopted for *hhl*, with axioms labeling the leaves of the tree, and inference figures creating the internal nodes. On the other hand, because of the limitations in expressive power of propositional CG, there are cases of extraction, like the non-peripheral ones, which cannot be handled so smoothly in Lambek Calculus; for instance, the sequent

$$NP \;\; ((S\backslash NP)/PP)/NP \;\; PP \;\Rightarrow\; S/NP$$

is not valid in the calculus, and this precludes a natural explanation of cases like

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

assuming that the verb **put** is assigned the type $((S\backslash NP)/PP)/NP$. Alternative solutions are possible, and one is explored in [37], but they involve quite a lot of complications from the point of view of lexical assignments.

These limitations in the expressive power of Lambek Calculus are reflected in the fact that it lacks all three of the structural rules of Interchange, Contraction and Thinning which characterize the Gentzen systems for Minimal, Intuitionistic and Classical Logic. This means that in Lambek Calculus:

- because of the lack of Interchange, premises have to be consumed in a fixed order
- because of the lack of Contraction, no premise can be used more than once

- because of the lack of Thinning, every premise must be used at least once

Indeed, lack of such structural rules provides proof-theoretically the power to enforce constraints on word order. [6]. Here below, we briefly examine how this compares with our framework.

Lack of Interchange versus Representation of String Positions

In Lambek Calculus, Interchange is not admitted, since the hypotheses used for deriving the type of a given string must also account for the positions of the words to which they have been assigned as types, and must obey the strict string adjacency requirement between functions and arguments of classical CG. Thus, Lambek Calculus must assume ordered lists of hypotheses, so as to account for word-order constraints; as recently explored in [4], relaxing this condition leads to a dramatic drop in the recognizing power of the calculus. By contrast, under our approach, word-order constraints are obtained declaratively, via sharing of string positions, and there is no strict adjacency requirement. In proof-theoretical terms, this directly translates in viewing programs as unordered sets of hypotheses.

Lack of Contraction versus Logical Variables

Lambek Calculus does not allow Contraction, so as to account for the fact that each type cannot identify more than one position in the input string. Thus, once we have used the type in a derivation, then we have also consumed the corresponding position in the input, and the type cannot be used anymore. We have seen how under our approach arguments of predicates in program clauses corresponding to string positions are logical variables at the beginning of the parsing process, and are fully instantiated terms at the end of it. Thus, no premise in the program can be used to individuate more than one constituent structure, and therefore we obtain the same effect, from the point of view of the output of the computation, achieved in Lambek Calculs by abolishing Contraction.

[6] For recent work investigating the proof-theoretic properties of Lambek Calculus, see [4, 38, 24].

Lack of Thinning versus Filtering of Vacuous Abstractions

Thinning is not admitted in Lambek Calculus to ensure that there are no unused hypotheses in a proof, so as to prevent ungrammatical cases such as

*which Paul shall put a book on the table

In our version of CG we can pass logical forms of parsed expressions as arguments to predicates. We have seen that cases like the one above result in logical forms corresponding to vacuous λ-abstractions, which can be ruled out via metalogical tests for non-vacuousness, in the same way as in the GPSG-style gap-introducing rules of Chapter 3.

The considerations above show that we can maintain all the structural rules, which are essential to preserve certain properties fundamental to the formal status of Hereditary Harrop Logic, like soundness and completeness with respect to Intuitionistic Logic, and to its usage as a general purpose logic programming language. On the other hand, the possibility of disallowing Thinning and Contraction over particulat kinds of premises would certainly be advantageous even from the point of view of the approach we are taking here. In fact, our formalization of CG presents the same problems we had for GPSG in the fact that unused premises corresponding to gaps must be controlled metalogically, by filtering proof trees encoded as vacuous abstractions. Moreover, here we are not fully exploiting the situation of *bounded computation* characterizing parsing with CG (in contrast, for instance, to parsing with GPSG), given by the fact that each premise is isomorphic with a corresponding occurrence of a word (possibly the empty word, in the case of gaps) in the input string, and thus can be used *exactly once* in the course of the computation. Clearly, Lambek Calculus is in better stand from this point of view, since lack of Thinning and Contraction will precisely produce the effect of making each premise usable exactly once in the course of a proof. In conclusion, the same considerations we made in Chapter 3 can be repeated here: a more flexible *hhl*-interpreter, drawing in spirit with

recent research in Linear Logic [13], would offer the ideal tool to solve the problems above, by being capable of distinguishing between *permanent* premises (i.e., premises subject to Thinning and Contraction) and *ephemeral* premises (i.e., premises which are not subject to Thinning and Contraction). We could thus maintain the deductive power needed for a general-purpose logic programming language, and the expressive power needed for the syntactic and semantic solutions that we have proposed for natural language analysis, while at the same time obtaining an elegant and fully logical way to control the computation.

5.5.2 Combinatory Grammar

Combinatory Grammar has been developed mainly by Steedman, Dowty and Szabolczi [1, 45, 46, 47, 48, 6], and is based on the use of certain *combinatory rules* which define a rewrite relation over sequences of bidirectional types. Chief among such rules are those of *Application* and *Composition*, the first one of which was already in use in Bar-Hillel's original formulation of bidirectional CG, while the other was the specific ingredient added to Bar-Hillel's framework in Ades and Steedman's paper [1], which historically started the trend of Combinatory Grammar; such a rule plays a crucial role in accounting for unbounded dependencies phenomena. As in Lambek Calculus, each rule comes in two versions, determined by the directionality of the types involved.

$X/Y \quad Y \longrightarrow X$	$Y \quad X\backslash Y \longrightarrow X$
(rightward application)	(leftward application)
$X/Y \quad Y/Z \longrightarrow X/Z$	$Y\backslash Z \quad X\backslash Y \longrightarrow X\backslash Z$
(rightward composition)	(leftward composition)

Another rule which plays an important role is that of *Type Raising*, which "raises" an argument into a function over its own function. This mapping is word-order preserving, in the sense that the raised function is of opposite

directionality of the function with respect to which it is raised.

$$X \longrightarrow Y/(Y\backslash X) \qquad X \longrightarrow Y\backslash(Y/X)$$

(rightward raising) (leftward raising)

The parsing problem in Combinatory Grammar has a rather different characterization from the way it can be expressed in the predicate logic framework we have presented here, or in Lambek Calculus, where it reduces to a form of theorem proving. The idea here is instead that of using the combinatory rules to obtain reductions in "normal form" of sequences of types. From a processing point of view, one of the main problems this approach needs to face is that there may be many different ways of achieving the same normal form, thus leading to redundant derivations for the same string. This problem has been baptized "spurious ambiguity", and some tentative solutions have been discussed in [40, 50] [7]. On the other hand, the proliferation of possible derivations for the same syntactic type can have its advantages, like the fact that it can model different intonation patterns for the same utterance, or that it can be exploited to provide attractive accounts of incremental processing, as in [14].

There is however an interesting relationship between Combinatory Grammar and proof-theoretic frameworks like the one presented here or Lambek Calculus, in that many of the combinatory rules correspond to theorems in such frameworks. (As far as our framework goes, we have of course to assume a non-directional version of such rules.) This explains the possibility of taking solutions of linguistic problems from Combinatory Grammar and transferring them to our version of CG, as was done above.

Let's take for instance the rule of Rightward Composition. It can be shown that for all bidirectional types X, Y and Z the sequent

$$X/Y \quad Y/Z \Rightarrow X/Z$$

[7] A more recent proposal in [16], which was not known at the time of writing the first draft of this dissertation, seems now to provide a very promising proof-theoretic direction for a full solution of the spurious ambiguity problem.

is a theorem of Lambek Calculus. An analogous result holds for Leftward Composition. An non-directional version of the same rule holds in Hereditary Harrop Logic, as shown in figure 5.5. Similarly, the rules of Application and of Type Raising can be derived as theorems. (Indeed, Application or, for that matter, Modus Ponens, is obtained as a subproof in both the proof for Composition and Type Raising.) We show in figure 5.6 a derivation of Rightward Raising in Lambek Calculus. In Hereditary Harrop Logic, a specific instance of such rule is given by the law of Double Negation, where negation is assumed to be defined in terms of implication and the distinguished propositional constant $\bot$ as

$$\sim P \equiv_{def} P \supset \bot$$

We use here the symbol $\sim$ for this kind of negation, to distinguish it from the negation $\neg$ used in the tests for the non-vacuousness of λ-abstractions, which corresponds to the well-known metalogical device of *negation by finite failure*, discussed in the next chapter. $\sim$ refers instead to a weaker form of negation, constructive in that demands the direct construction of the proof of a negative formula, which can be exploited to implement constraints in logical databases, as in [29]. In Figure 5.6, we give a proof for the sequent

$$P \;\Rightarrow\; \sim\sim P \equiv_{def} (P \supset \bot) \supset \bot$$

5.5.3 Categorial Unification Grammar and Polymorphism

The recent trend of Categorial Unification Grammar (CUG) [22, 49, 51] proposes the replacement of the propositional types of classical CG with record-like structures which can carry information about agreement, semantics, phonology etc. and can make use of variables to share information across substructures. Clearly, our approach fits neatly into this trend, and can at some extent be regarded a logical reconstruction of it. Indeed, a crucial point is that here the use of complex internal information is carried one fundamental step forward with respect to the previous attempts in CUG; in fact, we use it for the purpose of obtaining word-order constraints via sharing of string variables, while

$$\dfrac{\dfrac{\dfrac{X \Rightarrow X \qquad Y \Rightarrow Y}{X/Y \;\; Y \Rightarrow X}/-\textsf{left} \qquad Z \Rightarrow Z}{X/Y \;\; Y/Z \;\; Z \Rightarrow X}/-\textsf{left}}{X/Y \;\; Y/Z \Rightarrow X/Z}/-\textsf{right}$$

$$\dfrac{\dfrac{\dfrac{\{P \supset Q, Q \supset R, P\} \Rightarrow P}{\{P \supset Q, Q \supset R, P\} \Rightarrow Q}\,\textbf{(II)}}{\{P \supset Q, Q \supset R, P\} \Rightarrow R}\,\textbf{(II)}}{\{P \supset Q, Q \supset R\} \Rightarrow P \supset R}\,\textbf{(VII)}$$

Figure 5.5: Lambek Calculus proof for Rightward Composition (above) and *hhl* proof for Composition (below)

$$\dfrac{\dfrac{X \Rightarrow X \qquad Y \Rightarrow Y}{Y\backslash X \;\; X \Rightarrow X}\backslash-\textsf{left}}{X \Rightarrow Y/(Y\backslash X)}/-\textsf{right}$$

$$\dfrac{\dfrac{\{P, P \supset \bot\} \Rightarrow P}{\{P, P \supset \bot\} \Rightarrow \bot}\,\textbf{(II)}}{\{P\} \Rightarrow \sim\!\sim P \equiv_{def} (P \supset \bot) \supset \bot}\,\textbf{(VII)}$$

Figure 5.6: Lambek Calculus proof for Rightward Raising (above) and *hhl* proof for law of Double Negation (below)

CUG practitioners have so far been relying for this purpose on the use of the directional connectives of classical CG[8]. Of the arguments not corresponding to string positions, we have specifically focused on the ones accounting for logical forms, since there too we have introduced an element of novelty deriving from the use of generics in correlation with unbounded dependencies.

As we have explained in the Introduction, our effort here can be viewed as a form of reconciliation between the two trends within CG represented, respectively, by Lambek Calculus and Combinatory Grammar and by CUG. Practitioners of the first trend have been stressing the need of a powerful inferential and combinatorial machinery in order to account for complex natural language constructions, while they have been rather conservative on the structure of the objects on which such machinery operates; in general, they have used bare propositional types, and they have assumed that these could possibly be replaced with record-like structures to incorporate, say, information about agreement without significant changes in the overall theory. By contrast, people in the CUG trend have been stressing the importance of a rich representational structure, with the capability of sharing values across substructures, while they have been relying on very simple inferential operations, basically corresponding to the rule of Functional Application. But, as we have shown here, the proof-theoretic flavour from Lambek Calculus can be transferred to the fully standard logical context of one of the latest developments in Logic Programming, such that the inadequacies deriving from the poor representational structure of bidirectional propositional CG can be eliminated in virtue of the representational richness of predicate logic; at the same time, this allows us to take full advantage of the possibility of using quantified logic reasoning for syntactic and semantic processing.

Polymorphism

We conclude with some remarks about the realtionship between our approach to CG and polymorphism. A general claim behind the CUG trend is that it

[8]Indeed, Uszkoreit [49] mentions the possibility of replacing directional slashes with DCG-style string positions, but does not illustrate it further.

brings inside CG the idea of *polymorphic* type, in the same way as in functional programming languages like ML or in the polymorphic λ-Calculus. A polymorphic type, characterized by the occurrence of universally quantified variables in its structure, allows for *generic* functions, which can take values from different, possibly infinite monomorphic types. So, for instance, the function **append** can receive a type declaration such that it can be viewed as a function operating over lists of integers, or of strings, or of reals etc., while still maintaining a strongly typed kind of discipline.

In our version of CG, we have accounted for a very simple but very important kind of polymorphism, in the fact that a type assigned to a lexical item can be instantiated in terms of a potentially infinite number of string positions. However, we have seen that in the course of a proof such a polymorphism is realized boundedly, in the sense that we have only a finite number of instantiated occurrences of a given type in the program determined by a certain input string [9].

[9] In general, all kinds of quantified definite clauses as used in logic programming can be viewed under this perspective as polymorphic type declarations; an interesting approach to the formalization of this view of logic programming is given in [20].

Chapter 6

Implementation

6.1 Lambda Prolog

We rely for the actual computer implementation of the ideas presented in the previous chapters on the logic programming language λ-Prolog, a higher-order extension of Prolog. The current version 2.7 of λ-Prolog, developed by Dale Miller and Gopalan Nadathur at University of Pennsylvania [33], contains all the aspects of Hereditary Harrop Logic, with the exception of implications as goals. A new version, developed at Carnegie-Mellon University by Frank Pfenning and others, is expected soon; besides providing a general speed-up in terms of efficiency, this new version should also include implications as goals.

Thus, λ-Prolog in its final version will correspond to a full real-life incarnation of the abstract logic programming language of *hhl*, in the same way as standard Prolog is a real-life incarnation of Horn logic. For the time being, we have supplied the missing features through a meta-interpreter for *hhl* on top of the current version of λ-Prolog, which we illustrate briefly below.

6.1.1 A Metainterpreter for Hereditary Harrop Logic

In λ-Prolog notation, a λ-abstraction of the form

$$\lambda x P$$

is written as

```
X\P
```

A universally quantified goal is written as

```
(pi X\P)
```

and an existentially quantified one as

```
(sigma X\P)
```

λ-Prolog maintains the LISP-style notation

```
( ... )
```

to express combination of function and predicate symbols with arguments, and the ordinary Prolog notation for definite clauses

```
Head :- Body
```

Disjunction is expressed through the infix operator `;`.

In our metainterpreter, universal and existential quantification in the object language is expressed through the operators `all` and `some` in the form

```
(all X\P)
```

and

```
(some X\P)
```

Disjunction and conjunction are expressed through the infix operators `or` and `&`, and we mainatin the standard logical notation for definite clauses, in the form

```
Body ==> Head
```

However, we revert to Prolog notation in the fact that now variables begin with upper-case letters, and everything else with lower-case letters. Program clauses are stored in the database in the form

```
(rule Clause)
```

The metainterpreter is defined in terms of the two-place predicate **prove**. The first argument corresponds to a stack of program clauses added at run-time via the *AUGMENT* operation for implications as goal, or created by "program synthesis", as in the case of the two-step approach characterizing CG parsing. We assume however that there are also clauses which are part of the program "from the beginning"; although such clauses can in principle be stored in the first argument of **prove**, they are more efficiently accessed directly from the Prolog database via the predicate **rule**. The second argument of **prove** is the goal to be proved. We shall also make use of the distinguished predicate **truth**, which corresponds to an always succeeding goal.

We have then the following clauses:

```
prove Cl truth.

prove Cl (constraints Constraints) :- Constraints.

prove Cl (some G) :- sigma T\ (prove Cl (G T)).

prove Cl (all G) :- pi T\ (prove Cl (G T)).

prove Cl (G1 & G2) :-  prove Cl G1, prove Cl G2.

prove Cl (G1 or G2) :- prove Cl G1 ; prove Cl G2.

prove Cl (D ==> G) :-  conjuncts D Cs, append Cs Cl Cl0,
                       prove Cl0 G.

prove Cl A :-  backchain Cl A G, prove Cl G.

backchain [C|Cs] A G :-  instan C (G ==> A).

backchain [C|Cs] A G :- backchain Cs A G.
```

```
backchain Cl A G :- rule C, instan C (G ==> A).

instan (all P) C :- sigma T\ (instan (P T) C).

instan C C.

instan A (truth ==> A).

conjuncts (C1 & C2) [C1|Cs] :- conjuncts C2 Cs.

conjuncts C [C].
```

The behaviour of the metainterpreter can be explained, clause by clause, as follows.

Successful Termination

The clause

```
prove Cl truth.
```

defines successful termination, and implements proof rule **(I)**. In fact, the interpreter views an atomic program clause `A` as being really an implication of the form

```
truth ==> A
```

Thus, accessing of atomic clauses just becomes a special case of the backchaining rule **(II)** (as intuitively is), and successful termination simply amounts to the situation where we want to prove the always succeeding goal `truth`.

Escape to the Metalanguage

The clause

```
prove Cl (constraints Constraints) :- Constraints.
```

implies an escape of the execution at the level of the metalanguage, and allows to call metalogical predicates, like negation as finite failure, which is available in λ-Prolog. Thus, checking against vacuous abstraction can be expressed within a given clause as

```
(constraints (not (vacuous P)))
```

where `not` corresponds to negation as finite failure in λ-Prolog.

Quantifiers

The clause

```
prove Cl (some G) :- sigma T\ (prove Cl (G T)).
```

and

```
prove Cl (all G) :- pi T\ (prove Cl (G T)).
```

"absorb" universal and existential quantifier in the object language in terms of the corresponding quantifiers in the metalanguage, thus implementing proof rules **(V)** and **(VI)** for the *INSTANCE* and *GENERIC* search operation.

Conjunction and Disjunction

Similarly, the clauses

```
prove Cl (G1 & G2) :-  prove Cl G1, prove Cl G2.
```

and

```
prove Cl (G1 or G2) :- prove Cl G1 ; prove Cl G2.
```

absorb the definition of proof rules **(III)** and **(IV)** for the disjunction and conjunction connectives in terms of the corresponding connectives in the metalanguage, thus implementing the *AND* and *OR* search operations.

Implications as Goals

The clause

```
prove C1 (D ==> G) :-  conjuncts D Cs, append Cs C1 C10,
                       prove C10 G.
```

handles implications as goals, a feature which is still missing from the current version of λ-Prolog. What happens in this case is that the procedure `conjuncts` breaks the antecedent in the implication into a set of conjuncts, which then get appended to the stack of premises given by the first argument of the predicate `prove`. The fact that `prove` is called itself as a goal by the metainterpreter ensures that the logical variables which may be in the antecedent are preserved as such in the conjuncts which are added as program clauses. Thus, the nature of the augmentation of the program is local, and as soon as the branch of the computation involving the implication has been completely explored, then the program goes back to its previous status. This behaviour of the *AUGMENT* search operation can be contrasted with the behaviour of the Prolog predicate `assert`, which adds new clauses to the program globally and permanently[1].

Backchaining

We come now to the *Backchaining* search operation which is, in a sense, the workhorse of the interpreter. This is defined in terms of the following clause:

```
prove C1 A :-  backchain C1 A G, prove C1 G.
```

Here the `backchain` predicate finds a clause whose head matches the current goal, and then goes on trying to prove the body of the same clause. The search is done in terms of the `backchain` predicate which, as can be seen from its definition, searches both the temporary stack of clauses and the clauses permanently stored in the database.

[1]Indeed, whenever the use of `assert` in a Prolog program is meant to be local, then it can be logically accounted for in terms of the *AUGMENT* search operation, as shown in [29].

```
backchain [C|Cs] A G :-  instan C (G ==> A).

backchain [C|Cs] A G :- backchain Cs A G.

backchain Cl A G :- rule C, instan C (G ==> A).
```

The matching is done in terms of the **instan** predicate, which either instantiates with a new logical variable an explicitly quantified program clause, and then calls itself over the instantiation; or simply tries to unify a program clause with a clause having as head the current goal and as body an unspecified body; or makes **truth** the new goal in case the current goal unifies with an atomic clause. The use of explicit quantification is here crucial in allowing a fully universal reading of clauses added to the temporary stack, in that variables of clauses in the stack which are not explicitly quantified are logical variables, and as such cannot be instantiated more than once.

The three clauses below illustrate the three parts played by **instan**, in the order we have illustrated them above.

```
instan (all P) C :- sigma T\ (instan (P T) C).

instan C C.

instan A (truth ==> A).
```

6.1.2 Checking Vacuous Abstraction

We come now at the problem of checking vacuous abstraction of λ-expressions. This can be done straightforwardly in λ-Prolog, by exploiting the substitution laws for free variables in λ-Calculus. For a formal statement of such laws the reader is sent to [17]; however, the intuitive meaning of the substitution laws we want to exploit can be informally stated in the fact that no free variable can become captured by a variable-binding operator as a side-effect of substitution.

The desiderability of this condition can be appreciated from the fact that its violation would contradict certain natural assumptions about λ-terms. For instance, consider the term

$$\lambda y\, x$$

which corresponds to the constant function whose value is always x. If we substitute y for x in such a function, then the free variable y becomes bound as an effect of the substitution, and we have a completely different function with respect to the original one, given by the identity function

$$\lambda y\, y$$

This example gives us a direct hint on how to implement a check on the vacuousness of λ-terms. This can be obtained via the following definite clause:

```
vacuous(X\P)
```

Now, clearly any goal defined in terms of the predicate `vacuous` succeeds if and only if its argument is a vacuous λ-abstraction; for otherwise the body of the λ-expression in the definite clause above would be unifiable with something that involves a free variable `X` getting bound as an effect of the substitution involved in the unification. But this is not allowed in the unification algorithm behind λ-Prolog, since such an algorithm is defined in accordance with the substitution laws of λ-Calculus.

6.2 Parsing

We can use the notation developed for our implementation of *hhl* in λ-Prolog to make executable translations of the grammatical formalisms considered in the previous chapters. In the case of the GPSG-style DCGs of Chapter 3, parsing of a sentence just involves calling a goal of the form

```
(s Sentence [] LogicalForm)
```

with Sentence given. In the case of the DCGs with gap-filler information of chapter 4 parsing of a sentence involves instead calling a goal of the form

```
(s Sentence [] [] [] LogicalForm)
```

In the case of the CG-style approach developed in Chapter 5 we need instead to organize parsing in two steps, one to build a program from the types associated with the words in the input string, and the other to prove that the input string is a grammatical sentence This can be done in terms of the following set of clauses:

```
infer_type String Type :-
     program String Program,
     prove Program (s String [] LogicalForm).


program [Word] [P1] :- lex Word P,
                       instantiate_type P P1.

program [Word|Words] [P1|Q] :- lex Word P,
                               instantiate_type P P1,
                               program Words Q.


instantiate_type P P1 :- sigma T\ (instantiate_type (P T) P1).

instantiate_type P P.
```

Here the top-level predicate infer_type calls the two procedures program and prove implementing, respectively, the first and the second step of the parsing process. program simply loops through the string, building a program

from the types associated with the words in the input string. The predicate `instantiate_type`, called within `program` to instantiate the retrieved types with logical variables, is nothing but a variation of the predicate `instan` which is used by the *hhl*-metainterpreter to instantiate universally quantified program clauses. `prove` tries then to prove the grammaticality of the input string by deriving its type as a theorem from the built program.

Bibliography

[1] Ades, Anthony E. and Mark J. Steedman. 1982. *On the Order of Words.* Linguistics and Philosophy 4, pp517-588.

[2] Anderson, A. R. and Belnap, N. D. 1975. *Entailment* Volume 1. Princeton University Press, Princeton, New Jersey.

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

[4] van Benthem, Johan 1988. *The Lambek Calculus.* In R. Oehrle et al eds., *Categorial Grammars and Natural Language Structures*, Reidel, Dordrecht/Boston.

[5] Colmerauer, Alan 1978. *Metamorphosis Grammars.* In Leonard Bolc ed., *Natural Language Communication with Computers.* Springer-Verlag, Berlin, Germany.

[6] Dowty, David 1988. *Type Raising, Functional Composition, and Non-constituent Coordination* In R. Oehrle et al eds., *Categorial Grammars and Natural Language Structures*, Reidel, Dordrecht/Boston.

[7] Felty, Amy and Dale Miller 1988. *Specifying Theorem Provers in a Higher-order Logic Programming Language.* In Proceedings of the ninth International Conference on Automated Deduction, Argonne, Illinois

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

[9] Gazdar, Gerald. 1981. *Unbounded Dependencies and Coordinate Structure.* Linguistic Inquiry. 12. pp155-184

[10] Gazdar, Gerald, Ewan Klein, Geoffrey Pullum, Ivan Sag 1985. *Generalized Phrase Structure Grammar.* Blackwell's, Oxford, and Harvard University Press, Cambridge, Mass.

[11] Gazdar, Gerald, Geoffrey Pullum, Robert Carpenter, Ewan Klein, Thomas Hukari, Robert Levine 1988. *Category Structures.* In Computational Linguistics, vol.14, 1, pp1-20

[12] Gentzen, G. 1969. *Investigations in Logical Deduction.* In Szabo, M. E. (ed.) *The Collected Papers of Gerhard Gentzen.* North Holland, Amsterdam.

[13] Girard, J. Y. 1987. *Linear Logic.* Theoretical Computer Science. 50:1. pp1-102

[14] Haddock, N. 1988. Forthcoming Dissertation. University of Edinburgh.

[15] Harrop, R. 1960. *Concerning Formulae of the Types* $A \supset B \vee C$, $A \supset \exists x(B\ x)$ *in logical systems.* The Journal of Symbolic Logic, 25, pp27-32.

[16] Hepple, M. and G. Morrill 1989. *Parsing and Derivational Equivalence.* In Proceedings of the Association for Computational Linguistics, European Chapter, Manchester, UK.

[17] Hindley, Roger J. and Jonathan P. Seldin 1986. *Introduction to Combinators and λ-Calculus.* Cambridge University Press, Cambridge

[18] Howard, W. A. 1980. *The Formulae-as-Types Notion of Construction.* In Hindley, J. R. and J. P. Seldin (eds.) *To H. B. Curry, Essays on Combinatory Logic, Lambda Calculus, and Formalism.* Academic Press.

[19] Huet, Gerard *Unification in Typed Lambda-Calculus* in *Lambda-Calculus and Comp.Sci. Theory*, Springer Lecture Notes, 37.

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

[21] Schabes, Yves, Anne Abeille, and Aravind Joshi. 1988. *Parsing Strategies with Lexicalized Grammars: Application to Tree Adjoining Grammars.* In Proceedings of Coling88, Budapest.

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

[23] Klein, Ewan 1989. Personal Communication.

[24] König, Esther 1989. *Parsing as Natural Deduction.* In Proceedings of the 27th Annual Meeting of the Association for Computational Linguistics, Vancouver.

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

[26] Lloyd, J. and R. Topor. 1984. *Making Prolog More Expressive.* The Journal of Logic Programming. 1. pp225-240.

[27] McCarty, L. T. 1988 *Clausal Intuitionistic Logic I. Fixed Point Semantics.* The Journal of Logic Programming. 5. pp1-31.

[28] McCarty, L. T. 1988 *Clausal Intuitionistic Logic II. Tableau Proof Procedure.* The Journal of Logic Programming. 5. pp93-132.

[29] Miller, Dale. 1989. *A Logical Analysis of Modules in Logic Programming.* The Journal of Logic Programming. 6. pp79-108.

[30] Miller, Dale. 1989. *Lexical Scoping as Universal Quantification.* In Proceedings of the Sixth International Logic Programming Conference, Lisboa, Portugal.

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

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

[33] Miller, Dale and Gopalan Nadathur. 1988. *An Overview of λ-Prolog.* In Proceedings of the Fifth International Logic Programming Conference, Seattle.

[34] Miller, Dale, Gopalan Nadathur, Frank Pfenning and Andre Scedrov. 1989. *Uniform Proof Systems as a Foundation for Logic Programming* To appear in the Annals of Pure and Applied Logic.

[35] Montague, Richard. 1974. *Formal Philosophy* Yale University Press, New Haven, Connecticut.

[36] Morrill, Glyn and R. Carpenter 1989. *Compositionality, Implicational Logics and Theories of Grammars.* To appear in Linguistics and Philosophy.

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

[38] Moortgat, Michael. 1988. *Categorical Investigations.* Foris Publications, Dordrecht, Holland.

[39] Pareschi, Remo 1988. *A Definite Clause Version of Categorial Grammar.* In Proceedings of the 26th Annual Meeting of the Association for Computational Linguistics, University of Buffalo.

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

[41] Pereira, Fernando C. 1981 *Extraposition Grammars.* Computational Linguistics, vol 7, 4, pp243-256.

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

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

[44] Pollard, Carl J. 1988. *Categorial and Phrase Structure Grammar.* In R. Oehrle et al eds., *Categorial Grammars and Natural Language Structures*, Reidel, Dordrecht/Boston.

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

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

[47] Steedman, Mark J. 1988. *Combinators and Grammars.* In R. Oehrle et al eds., *Categorial Grammars and Natural Language Structures*, Reidel, Dordrecht/Boston.

[48] Szabolczi, Anna 1987. *Bound Variables in Syntax: are there any?* Paper presented at the sixth Amsterdam Colloquium.

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

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

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