Indexing Methods for Efficient Parsing with Typed Feature Structure Grammars

Cosmin Munteanu

A thesis submitted in conformity with the requirements

for the degree of Master of Science

Department of Computer Science

University of Toronto

mcosmin@cs.toronto.edu

Abstract

A major obstacle in developing efficient parsers for unification-based grammars is the slow parsing time caused by the complex and large structure used to represent grammatical categories. With the broadening coverage of such grammars, their size and complexity increases, creating the need for improved parsing techniques. Although several statistical optimizations exist today that exhibit significant improvements in parsing times, they rely on data collected during a training phase. This thesis presents a theoretical investigation of indexing based on static analysis of feature structure grammar rules, a method that has received little attention in the last few years in computational linguistics. This non-statistical approach has the advantage of not requiring training phases, although it is consistent with further statistical optimizations.

Contents

List of Figures **ix**

List of Tables **xi**

1 Introduction **1**

1.1 Overview 1

1.2 Motivation 2

1.3 Contributions 3

1.4 Structure of the Thesis 4

2 Typed Feature Structures **7**

2.1 Type Hierarchies 8

2.2 Typed Feature Structure Definition 9

2.2.1 Example 11

2.3 Typed Feature Structures Operations 12

2.3.1 Subsumption 12

2.3.2 Unification 14

2.3.3 Descriptions and Most General Satisfiers 16

2.4 Extensions 17

2.4.1 Appropriateness 17
2.4.2 Well-Typed Feature Structures 18
2.4.3 Statically Typable Signatures 19
2.4.4 Structure Sharing 20
2.5 Typed Feature Structure Cuts 22

3 Unification-Based Grammars **25**
3.1 Head-Driven Phrase Structure Grammars 27
3.1.1 Principles of HPSG 27
3.1.2 The HPSG Formalism 28
3.2 Implementation Aspects 31
3.2.1 Compiling Grammar Rules 31
3.2.2 Variables in TFS Representations 34
3.2.3 TFS Grammar Rules 38
3.2.4 Encoding of TFSs 40

4 Indexing for TFSG Parsing **45**
4.1 Chart Parsing with Typed Feature Structure Grammars . . . 46
4.1.1 Parsing with Typed Feature Structures 46
4.1.2 Bottom-Up Parsing 47
4.1.3 EFD Parsing 48
4.1.4 Parsing as Unification of MRSs 50
4.2 The Typed Feature Structure Indexing Problem 52
4.2.1 Difficulties for TFSG Indexing 52
4.2.2 Indexing Timeline 53
4.3 Indexed Chart Parsing 54

4.3.1 General Indexing Strategy 55
4.3.2 Using the Index 56
4.3.3 An Example 56
4.4 Previous Approaches to Indexing and Filtering TFSGs 58
4.4.1 HPSG Parsing with CFG Filtering 58
4.4.2 Quick Check 59
4.4.3 Rule Filtering 61
4.5 A Discussion About Optimization Approaches 62
4.5.1 Indexing vs. Filtering 63
4.5.2 Statistical vs. Non-Statistical Methods 64

5 Indexing for Non-Parsing Applications 67
5.1 Indexing For Other TFS Applications 67
5.1.1 An Indexing Scheme for TFS Retrieval 68
5.1.2 Automaton-based Indexing for Lexical Generation . . . 68
5.2 General Term Indexing 69
5.2.1 Attribute-Based Indexing 70
5.2.2 Set-Based Indexing 70
5.2.3 Tree-Based Indexing 71
5.3 Indexing in Database Systems 72

6 TFSG Indexing through Static Analysis 75
6.1 Positional Indexing 76
6.1.1 Building the Index 76
6.1.2 Using the Index 78
6.2 Path Indexing 78

6.2.1 Static Analysis of Grammar Rules 79
6.2.2 Building the Path Index 93
6.2.3 Key Extraction in Path Indexing 94
6.2.4 Using the Path Index 95

7 Experimental Evaluation **97**
7.1 Resources 97
7.2 Prolog Data Structure 99
7.3 Experiments 102
7.3.1 Evaluation using the unconstrained MERGE 102
7.3.2 Evaluation using the constrained MERGE 104
7.3.3 Comparison between statistical and non-statistical optimizations 106
7.4 Evaluation on Other UBGs 111
7.4.1 The Alvey Grammar 111
7.4.2 Penn Treebank CFG 113

8 Conclusions and Future Work **119**
8.1 Conclusions 119
8.2 Future Work 120

Bibliography **123**

List of Figures

2.1 A simple type hierarchy. 12
2.2 A simple typed feature structure, represented as an AVM. . . 12
2.3 A simple typed feature structure, represented as a directed graph. 13
2.4 A subsumption example. 14
2.5 Structure sharing in a graph representation of a TFS. 20
2.6 Structure sharing in an AVM representation of a TFS. 21
2.7 The unification between a TFS and a cut TFS. 23

3.1 A Simple HPSG 30
3.2 A phrase rule in ALE. 33
3.3 Structure sharing between two typed feature structures ("external sharing"). 36
3.4 Structure sharing between two typed feature structures, after unification. 37
3.5 A phrase rule seen as a MRS. 41

4.1 A simple example of indexed chart parsing. 57
4.2 Quick check vectors 61

6.1 Static Cut – An Example. 82

6.2 An example of the applicability of Proposition 6.3. 87

6.3 An example of Proposition 6.3 limitation to $|[\widehat{x}]_{\bowtie} \cap \widehat{M}| = 1$. . 87

6.4 An example of nodes in Case C of the proof for Proposition 6.6. 92

7.1 The encoding of TFSs as Prolog terms 100

7.2 The encoding of the TFS for the word *mary* as Prolog terms . 101

7.3 Parsing times for EFD, EFD with positional indexing, and EFD with path indexing applied to the unconstrained MERGE grammar. 103

7.4 Parsing times for EFD, EFD with positional indexing, and EFD with path indexing applied to the constrained MERGE grammar. 105

7.5 Parsing times for EFD and EFD with quick-check applied to the unconstrained MERGE grammar. The sentence numbers are the same as those used in Figure 7.3. 108

7.6 Parsing times for EFD and EFD with quick-check applied to the constrained MERGE grammar. The sentence numbers are the same as those used in Figure 7.4. 109

7.7 Parsing times for EFD and EFD-indexing applied to the Alvey grammar. 112

7.8 Parsing times for EFD and EFD-indexing applied to CFGs with atomic categories. 116

List of Tables

7.1 The number of successful and failed unifications for the non-indexed and indexed parsers over the unconstrained MERGE grammar. 104

7.2 The number of successful and failed unifications for the non-indexed and indexed parsers over the constrained MERGE grammar. 106

7.3 The set-up times for non-statistically indexed parsers and statistically optimized parsers for MERGE gramamr. 107

7.4 The number of successful and failed unifications for the non-indexed, path-indexed, and quick-check parsers over the unconstrained MERGE grammar. 110

7.5 Successful unification rate for the non-indexed CFG parser. . . 114

7.6 The grammars extracted from the Wall Street Journal directories of the PTB II. 115

Chapter 1

Introduction

1.1 Overview

Developing efficient parsers is one of the long-standing goals of research in natural language processing. A particular area of grammar development in strong need of improvements in parsing times is that of typed feature structure grammars (TFSGs). With respect to parsing times, much simpler grammar formalisms such as context-free grammars (CFGs) also face the problem of slow parsing time when the size of the grammar increases significantly. While TFSGs are small compared to large-scale CFGs (in terms of the number of rules), the problematic parsing times are generated by the complex structure required to represent the categories in the grammar rules. For example, in HPSGs [Pollard and Sag, 1994] covering the English language, one category could incorporate thousands of feature values (while in CFGs, the categories are atomic).

For TFSG chart parsers, one of the most time-consuming operations is the retrieval of categories from the chart. This is a look-up process: the retrieved category should match a daughter description from the grammar. The size of the typed feature structures [Carpenter, 1992], combined with their complex structure, results in slow unification (matching) times, which leads to slow parsing times. Therefore, while retrieving categories from the chart, failing unifications should be avoided. Using indexing methods to search for categories in the chart is also motivated by the successful use of indexing in the retrieval/updating process in databases[1].

1.2 Motivation

Most of the research aimed at improving the times of the retrieval component of parsing uses statistical methods that require training. During grammar development, the time spent for the entire edit-test-debug cycle is important. Therefore, a method that requires considerable time for gathering statistical data by parsing a training corpus could burden the development process. Indexing methods that are time efficient for the entire grammar development cycle are a necessary alternative.

Current techniques (such as the "quick-check", [Malouf *et al.*, 2000]) reduce the parsing times by filtering out unnecessary unifications based on statistically derived filters. Widely used in databases [Elmasri and Navathe, 2000] and automated reasoning [Ramakrishnan *et al.*, 2001],

[1]Indexing is one of the most popular research topics in databases. Presentations of existing indexing methods can be found in the vast database literature, from introductory textbooks ([Elmasri and Navathe, 2000]) to advanced research topics ([Bertino *et al.*, 1997; Manolopoulos *et al.*, 1999]).

indexing presents the advantage of a more principled, yet flexible, approach. Compared to simple filtering, an indexing structure allows for searches based on multiple criteria. It also provides support for efficiently processing complex queries that are not limited to membership checking [Manolopoulos *et al.*, 1999], such as range queries (only when the index is organized as a tree, not as a hash). The flexibility is reflected in the ease of adapting an existing parser to different grammars: an indexed parser needs only changes in the search criteria (a process that can be entirely automated), while a statistically optimized parser needs re-training.

1.3 Contributions

Indexing as a solution to improving parsing times for typed feature structure grammars is a research topic that has received little attention (and no direct attention for parsing) in the last few years in computational linguistics. This thesis is an argument in favour of the worthiness of deeper research in this area. The following contributions of this thesis prove the viability of such an argument:

Overview of existing work. A review of the existing research on improving the retrieval component of TFSG parsing is conducted. The extensive investigation of literature on this topic identifies the strengths and weaknesses of existing methods; it reveals the lack of an approach to indexing, as well as of a thorough analysis of the grammar rules in TFS-based parsers that can lead to the development of more efficient parsers.

Theoretical investigations. A formalization of the static analysis of grammar rules for parsing is presented. A non-statistical approach to indexing TFS-based parsers is developed based on the static analysis. This approach does not preclude further statistical optimizations.

Preliminary evaluation. The proposed indexing methods are integrated in a Prolog-implemented parser. A preliminary evaluation is conducted using a typed feature structure grammar.

1.4 Structure of the Thesis

The rest of this thesis is structured as follows:

Chapter 2: Typed Feature Structures presents a short introduction to typed feature logic. The main notations and definitions are outlined, while several useful extensions are introduced.

Chapter 3: Unification-Based Grammars an important grammatical formalism, Head-driven Phrase Structure Grammars, belonging to the class of unification-based grammars is presented here. Implementational aspects particular to Typed Feature Structure Grammars are also outlined. A new classification of the instances of variables used in TFSGs is introduced.

Chapter 4: Indexing for TFSG Parsing presents the general problem of parsing with TFSGs, the motivation and details of indexing the chart for such grammars. A general indexing strategy suitable for chart-based parsers is introduced. The chapter concludes with a review of existing techniques that are similar to the proposed indexing method, discussing about the differences between filtering (as done by the existing methods) and indexing, as well as between statistical and non-statistical approaches, are discussed.

Chapter 5: Indexing for Non-Parsing Applications investigates the current indexing research in related areas, such as information retrieval and lexicon indexing for generation. An overview of indexing for automated reasoning and databases is also presented.

Chapter 6: TFSG Indexing through Static Analysis is the core chapter of this thesis. It introduces a simple indexing method (positional indexing), followed by the presentation of the theoretical foundations for our static analysis of grammar rules. A complete indexing strategy based on the static analysis is then introduced. The chapter concludes with the presentation of a practical indexing scheme (path indexing) derived from the static analysis.

Chapter 7: Experimental Evaluation presents a preliminary evaluation of the indexing methods proposed in this thesis using a wide-coverage TFSG, followed by an investigation of the applicability of the proposed indexing methods to non-TFS grammar formalisms.

Chapter 8: Conclusions and Future Work outlines the main achievements and proposes directions for future work.

Chapter 2

Typed Feature Structures

Typed feature structures (TFS) are widely used in natural language processing and computational linguistics to enrich syntactic categories. They are very similar to frames from knowledge representation systems or records from various programming languages like C.

The motivation for employing TFSs in areas of computational linguistics such as grammar development can easily be observed through the following example [Allen, 1994]: in order to correctly parse the English phrase "a man" with context-free grammars (CFGs), the general rule

`NP` $\rightarrow$ `ART N`

is not sufficient (allowing for parses of the incorrect phrase "a men"). Instead, two rules:

`NP-sg` $\rightarrow$ `ART-sg N-sg`

`NP-pl` $\rightarrow$ `ART-pl N-pl`

are needed to capture the correct number agreement. Moreover, additional information would increase the size of the grammar. Feature structures represents a solution by embedding this kind of information inside more complex categories:

$$\begin{bmatrix} np \\ NUMBER: \quad N \end{bmatrix} \rightarrow \begin{bmatrix} art \\ NUMBER: \quad N \end{bmatrix} \begin{bmatrix} n \\ NUMBER: \quad N \end{bmatrix}$$

Sections 2.1 to 2.4 of this chapter will present a formal description of typed feature structures, following the formalism and notation of [Carpenter, 1992]. The reader familiar with this formalism is invited to move directly to the last section (2.5) of this chapter, where a useful addition to feature structure operations, the feature structure cut, is introduced.

2.1 Type Hierarchies

While feature structures are used to organize knowledge, types present "an additional dimension along which to classify or organize knowledge" [Penn, 2000]. In connection to feature structures, types can be seen as organizing feature structures into classes [Carpenter, 1992].

Types are organized into partially ordered sets, called type hierarchies. A complete definition of type hierarchies is given in [Carpenter, 1992] and [Penn, 2000]; in this section, only the definitions and theorems relevant to this thesis are presented.

Definition 2.1. *A partial order on a set P is a relation $\leq \subseteq P \times P$, such that $\forall x, y, z \in P$:*

- $x \leq x$ *(reflexivity),*
- *if* $x \leq y$ *and* $y \leq x$, *then* $x = y$ *(anti-symmetry),*

- *if $x \leq y$ and $y \leq z$, then $x \leq z$ (transitivity).*

Definition 2.2. *Given a partially ordered set $\langle P, \leq \rangle$, the set of upper bounds of a subset $S \subseteq P$ is the set $S^u = \{y \in P | \forall x \in S, x \leq y\}$.*

Definition 2.3. *A partially ordered set $\langle P, \leq \rangle$ is bounded complete iff, $\forall S \subseteq P$ such that $S^u \neq \emptyset$, S^u has a least element, called the least upper bound, or join, of S, written $\bigvee S$.*

Definition 2.4. *A type hierarchy is a non-empty, countable, bounded complete, partially ordered set.*

For the particular case of interest to typed feature structures, three points must be noted. First, the order relation between types is subtyping ($\sqsubseteq$), therefore a type hierarchy will be denoted as $\langle Type, \sqsubseteq \rangle$. Second, the least upper bound of a set S ($\bigvee S$) is written $\bigsqcup S$ (when S consists of only two elements x and y, $\bigsqcup S = x \sqcup y$ – x and y are said to unify, and $\sqcup$ is the type unification operation). Finally, there is a special least element for non-empty bounded complete partial orders of types: $\bot$ (“bottom” – the least type, or the most general type, which is more general than all other types in the hierarchy).

2.2 Typed Feature Structure Definition

For a given type hierarchy $\langle Type, \sqsubseteq \rangle$, and a finite set of features $Feat$, a typed feature structure can be defined as a rooted graph, where arcs are labeled with feature names and nodes are being labeled with types (feature values) [Carpenter, 1992; Penn, 2000]:

Definition 2.5. *A typed feature structure over Type and Feat is a tuple* $F = \langle Q, \overline{q}, \theta, \delta \rangle$ *where:*

- Q *is a finite set of nodes,*
- $\overline{q} \in Q$ *is the root node,*
- $\theta : Q \to Type$ *is a total node typing function,*
- $\delta : Feat \times Q \to Q$ *is a partial feature value function*

such that $\forall q \in Q$, $\exists$ *a finite sequence of features* $f_1, \ldots, f_n \in Feat$ *that connects* $\overline{q}$ *to* q *with* δ: $q = \delta(f_n, \ldots \delta(f_2, \delta(f_1, \overline{q})))$.

The feature value function can be extended to paths. A path is a sequence of features [Carpenter, 1992; Penn, 2000], and the feature value function δ applied to a path π and a node q gives the node reached by following π from q.

Definition 2.6. *Given the set of all features Feat, a path is a finite sequence of features,* $\pi \in Feat^*$.

The set of all paths will be referred to as $Path$.

Definition 2.7. *Given a typed feature structure* $F = \langle Q, \overline{q}, \theta, \delta \rangle$*, its (partial) path value function is* $\delta' : Feat^* \times Q \to Q$ *such that:*

- $\delta'(\epsilon, q) = q$,
- $\delta'(f\pi, q) = \delta'(\pi, \delta(f, q))$.

Definition 2.8. *Given a typed feature structure* $F = \langle Q, \overline{q}, \theta, \delta \rangle$*, if* $\delta(\pi, q)\downarrow$*, then the restriction of* F *to* π *is* $F@\pi = \langle Q', \overline{q'}, \theta', \delta' \rangle$ *where:*

- $\overline{q'} = \delta(\pi, \overline{q})$,
- $Q' = \{\delta(\pi', \overline{q'}) | \pi' \in Path\}$,
- $\delta'(f, q) = \delta(f, q)$ *if* $q \in Q'$,
- $\theta'(q) = \theta(q)$ *if* $q \in Q'$.

Definition 2.9. *In a typed feature structure with the root* $\overline{q}$ *and the feature value function* δ*, the node* x *is the ancestor of a node* x' *iff* $\exists \pi, \pi' \in Path$ *such that* $\delta(\pi, \overline{q}) = x$ *and* $\delta(\pi', x) = x'$.

2.2.1 Example

A simple type hierarchy, with types needed to define the agent and the object of the action "throwing", is presented in Figure 2.1. A typed feature structure, describing a particular instance of the action "throwing", in which the agent ("thrower") is of type **index**, **third** person, **masculine**, and **singular**, and the object ("thrown") is of the same type, has the same person and number, but different gender, is presented in Figure 2.2. The types (feature values) are written in boldface lowercase, while the features (THROWER, THROWN, PERSON, NUMBER, GENDER) are in regular uppercase letters. The representation used in Figure 2.2 is known as an *attribute-value matrix* (AVM), but typed feature structures can also be represented as labeled directed graphs, as in Figure 2.3.

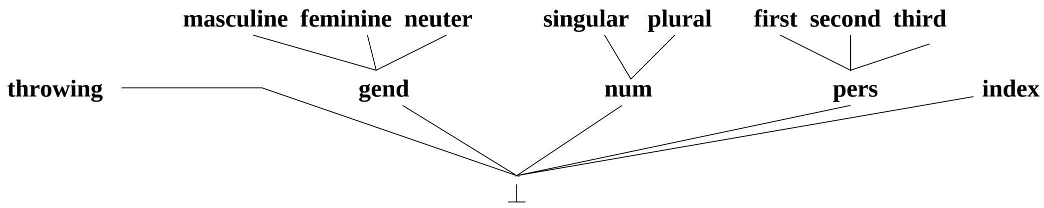


Figure 2.1: A simple type hierarchy.

$$\begin{bmatrix} \textbf{throwing} & \\ \text{THROWER:} & \begin{bmatrix} \textbf{index} & \\ \text{PERSON:} & \textbf{third} \\ \text{NUMBER:} & \textbf{singular} \\ \text{GENDER:} & \textbf{masculine} \end{bmatrix} \\ \text{THROWN:} & \begin{bmatrix} \textbf{index} & \\ \text{PERSON:} & \textbf{third} \\ \text{NUMBER:} & \textbf{singular} \\ \text{GENDER:} & \textbf{neuter} \end{bmatrix} \end{bmatrix}$$

Figure 2.2: A simple typed feature structure, represented as an AVM.

2.3 Typed Feature Structures Operations

Several operations with typed feature structures can be defined. Three of them are of interest for the work presented in this thesis: subsumption, unification, and most general satisfier.

2.3.1 Subsumption

Typed feature structure subsumption is an extension of the order relation between types. It defines a relation between feature structures that express the fact that one feature structure is more informative than the other.

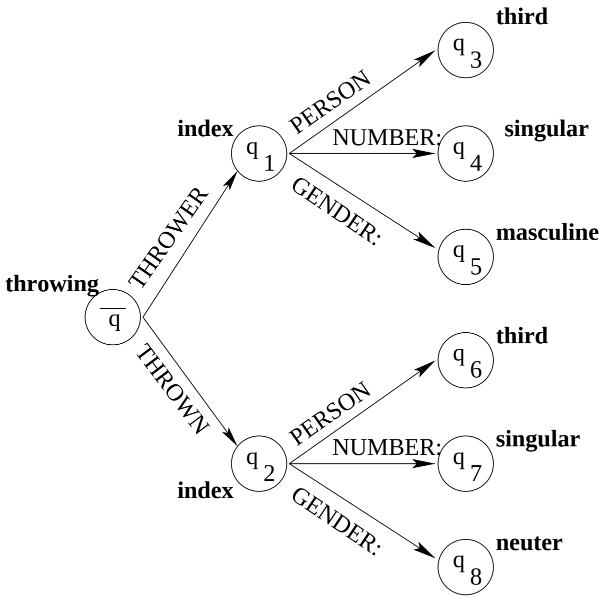


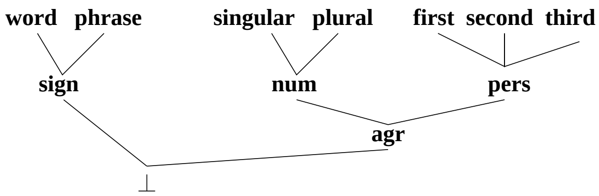
Figure 2.3: A simple typed feature structure, represented as a directed graph.

Formally, this is again defined in [Carpenter, 1992]:

Definition 2.10. *For a type hierarchy* $\langle Type, \sqsubseteq \rangle$ *and a finite set of features* $Feat$*, a feature structure* $F = \langle Q, \overline{q}, \theta, \delta \rangle$ *subsumes a feature structure* $F' = \langle Q', \overline{q}', \theta', \delta' \rangle$, $F \sqsubseteq F'$*, iff there is a morphism* $h : Q \rightarrow Q'$ *such that:*

- $h(\overline{q}) = \overline{q}'$,
- $\forall q \in Q,\ \theta(q) \sqsubseteq \theta'(h(q))$,
- $\forall f \in Feat, \forall q \in Q$, *such that* $\delta(f, q)$ *is defined,* $h(\delta(f, q)) = \delta'(f, h(q))$.

Figure 2.4 illustrates an example of a subsumption relation between two feature structures [Carpenter, 1992], given a type hierarchy. An interesting observation is that, computationally, subsumption checking is not expensive, being computed in linear time of the size of the feature structure.



$$\begin{bmatrix} \textbf{sign} \\ \text{AGR: } \begin{bmatrix} \textbf{agr} \\ \text{PERS: } \textbf{first} \end{bmatrix} \end{bmatrix} \sqsubseteq \begin{bmatrix} \textbf{phrase} \\ \text{AGR: } \begin{bmatrix} \textbf{agr} \\ \text{PERS: } \textbf{first} \\ \text{NUM: } \textbf{singular} \end{bmatrix} \end{bmatrix}$$

Figure 2.4: A subsumption example.

2.3.2 Unification

The unification of two typed feature structures is an operation that produces a feature structure that is their least upper bound, with respect to the subsumption ordering. Unification fails when the two feature structures provide inconsistent information.

Intuitively, unification is accomplished by first unifying the types of the root nodes of the two feature structures, and replacing them with the result of the type unification. By recursion, nodes that are values of identical features are then unified, and replaced with the result of the unification. Failure occurs when an inconsistency between nodes occurs (i.e., when two types can not unify).

In order to formally describe unification, the notion of alphabetic variants must be defined:

Definition 2.11. *Two feature structures are alphabetic variants, $F \sim F'$, if $F \sqsubseteq F'$ and $F' \sqsubseteq F$.*

Based on the definition of alphabetic variants, as well as on the properties of equivalence relations (transitive, reflexive, and symmetric), equivalence classes over a set X ($[x]_\equiv = \{y \in X | y \equiv x\}$), and the quotient set of a set X modulo $\equiv$ ($X/_\equiv = \{[x]_\equiv | x \in X\}$), Carpenter [1992] defines feature structure unification as:

Definition 2.12. *Suppose F, F' are feature structures such that $F \sim \langle Q, \overline{q}, \theta, \delta \rangle$, $F' \sim \langle Q', \overline{q}', \theta', \delta' \rangle$, and $Q \cap Q' = \emptyset$. The equivalence relation $\bowtie$ on $Q \cup Q'$ is defined as:*

- $\overline{q} \bowtie \overline{q}'$,
- *if $\delta(f, q)$ and $\delta'(f, q')$ are defined, and $q \bowtie q'$, then $\delta(f, q) \bowtie \delta'(f, q')$.*

The unification of F and F' is then defined as:

$$F \sqcup F' = \langle (Q \cup Q')/_{\bowtie}, [\overline{q}]_{\bowtie}, \theta^{\bowtie}, \delta^{\bowtie} \rangle$$

where

$$\theta^{\bowtie}([q]_{\bowtie}) = \bigsqcup \{(\theta \cup \theta')(q') | q' \bowtie q\}$$

and

$$\delta^{\bowtie}(f, [q]_{\bowtie}) = \begin{cases} [(\delta \cup \delta')(f, q)] \bowtie, & \textit{if } (\delta \cup \delta')(f, q) \textit{ is defined} \\ \textit{undefined}, & \textit{otherwise} \end{cases}$$

if all the joins in the definition of $\theta^{\bowtie}$ exist. $F \sqcup F'$ is undefined otherwise.

Computationally, unification has a complexity of the order of $O(ack^{-1}(n) \cdot n)$, where n is the size of the feature structure and ack is the Ackermann's function. However, as it will be shown in Section 2.4.2, the extensions needed in many applications increase the cost of the unification.

2.3.3 Descriptions and Most General Satisfiers

Descriptions are used to logically describe specific feature structures. While AVMs can be used to graphically describe feature structures, descriptions can be seen as a "way to talk about feature structures" [Carpenter, 1992] through the use of a logical language. This section provides a short summary of the complete presentations of descriptions and satisfiability found in [Carpenter, 1992] and [Penn, 2000].

Definition 2.13. *The set of descriptions, over a set of types Type, set of variables Vars, and set of features Feat, is the least set Desc such that:*

- $\sigma \in Desc$*, for all* $\sigma \in Type$*,*
- $x \in Desc$*,* $\forall x \in Vars$*,*
- $\pi : \phi \in Desc$*, for all paths* $\pi \in Feat^*$ *and* $\phi \in Desc$*,*
- $\pi \doteq \pi' \in Desc$*, if* $\pi, \pi' \in Desc$*,*
- $\phi \wedge \psi \in Desc$*, if* $\phi \in Desc$ *and* $\psi \in Desc$*.*

Definition 2.14. *The satisfaction relation* $\models$ *between feature structures and descriptions is the least relation such that:*

- $F \models \sigma$ *if* $\sigma \in Type$ *and* $\sigma \sqsubseteq \theta(\overline{q})$*,*
- $F \models \pi : \phi$ *if* $F@\pi \models \phi$*,*
- $F \models \pi \doteq \pi'$ *if* $\delta(\pi, \overline{q}) = \delta(\pi', overlineq)$*,*
- $F \models \phi \wedge \psi$ *if* $F \models \phi$ *and* $F \models \psi$*,*

While in [Carpenter, 1992] disjunctions are used in the description language, in this thesis, the description language has no disjunction defined. One can consider a disjunction in a description as specifying two distinct descriptions.

Every satisfiable description is logically satisfied by a most general feature structure. This most general feature structure of a description ϕ is called $MGSat(\phi)$, and $MGSat(\phi) \sqsubseteq F$, for every F that satisfies ϕ. The Most General Satisfier provides a way of mapping a description to a feature structure.

2.4 Extensions

Several extensions can be defined to augment the simple typed feature structure formalism presented so far. Without these extensions, typed feature structures can have features with arbitrary values [Carpenter, 1992]; defining several restrictions can enhance the logical control over non-determinism.

2.4.1 Appropriateness

Actual systems using type feature structures, such as ALE [Carpenter and Penn, 2001], require a strict specification of the possible types a feature value can have and of the possible types a feature can be defined on. This strict specification is known as an *appropriateness specification.* Appropriateness was first proposed by Pollard and Sag [1987] in order to differentiate between cases where there is a lack of information about the value of a feature, and cases where that feature is irrelevant. The formal definition of appropriateness is given in [Carpenter, 1992]:

Definition 2.15. *An appropriateness specification over the type hierarchy* $\langle Type, \sqsubseteq \rangle$ *and features Feat is a partial function* $Approp : Feat \times Type \rightarrow Type$ *such that:*

- $\forall F \in Feat$, $\exists$ *a most general type* $Intro(F)$ *for which* $Approp(F, Intro(F))$ *is defined (feature introduction),*
- *if* $Approp(F, \sigma)$ *is defined and* $\sigma \sqsubseteq \tau$*, then* $Approp(F, \tau)$ *is also defined and* $Approp(F, \sigma) \sqsubseteq Approp(F, \tau)$ *(upward closure).*

The appropriateness specification cannot be treated independently of the type hierarchy, and their specification together is known as the type signature [Penn, 2000]:

Definition 2.16. *A type signature is a structure* $\langle Type, \sqsubseteq, Feat, Approp \rangle$*, where* $\langle Type, \sqsubseteq \rangle$ *is a type hierarchy, Feat is a finite set of features, and Approp is an appropriateness specification.*

2.4.2 Well-Typed Feature Structures

Appropriateness itself does not specify how type and feature restrictions are interpreted. There are several choices of interpreting the appropriateness specification, ranging from very relaxed to more severe constraints placed on types and feature values.

In a basic interpretation, the appropriateness specification can be seen as a restriction on which features a type can bear, i.e., which labels are permitted for the arcs emanating from a node [Penn, 2000]. A much stronger restriction (well-typedness) places constraints on the type of value those features can have [Carpenter, 1992]:

Definition 2.17. *A feature structure $F = \langle Q, \overline{q}, \theta, \delta \rangle$ is well-typed iff $\forall q \in Q$, if $\delta(f,q)$ is defined, $Approp(f, \theta(q))$ is defined, and $Approp(f, \theta(q)) \sqsubseteq \theta(\delta(f,q))$.*

Well-typedness does not impose any restrictions on the existence of features. However, in TFS applications, such as ALE [Carpenter and Penn, 2001], it is practical to require that all features are defined (given values) for nodes for which they are appropriate. This condition, named total well-typedness, also includes the restrictions imposed by well-typedness [Carpenter, 1992]:

Definition 2.18. *A feature structure $F = \langle Q, \overline{q}, \theta, \delta \rangle$ is totally well-typed iff it is well-typed and $\forall q \in Q$ if $Approp(f, \theta(q))$ is defined, then $\delta(f,q)$ is defined.*

Imposing total well-typedness creates a distinction between absent information and irrelevant information inside a feature structure [Penn, 2000]. The work presented in this thesis is based on the assumption of totally well-typedness.

2.4.3 Statically Typable Signatures

An important aspect with respect to the implementation of typed feature structures is the static typing of type systems. A type system is static if all type inference and checking can be performed at compile-time [Carpenter, 1992]. The advantage of static typing resides in the increased efficiency of feature structure implementation, an advantage also seen in programming languages, where the most efficient languages, like C, require static typing of their type systems.

For typed feature structures, static typing is ensured when unification of well-typed feature structures results in a well-typed feature structure.

2.4.4 Structure Sharing

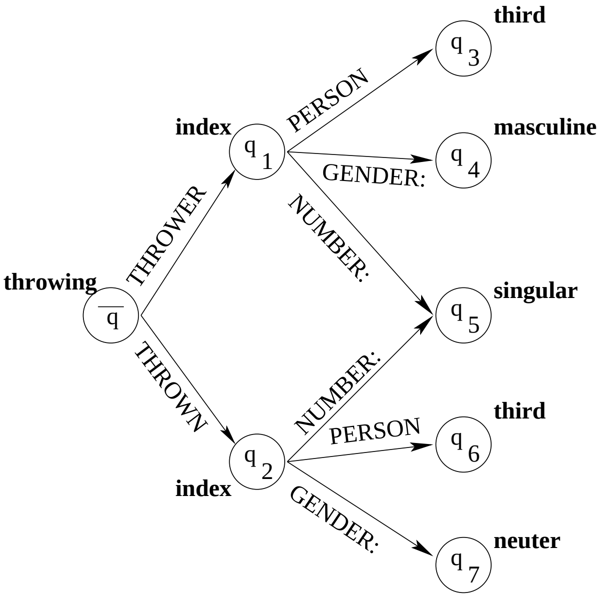


Figure 2.5: Structure sharing in a graph representation of a TFS.

From the long list of possible extensions to the basic formalism of typed feature structures, one of practical importance will be described before concluding this chapter. As seen in the example presented in Figure 2.3, both feature THROWER and feature THROWN have as value a feature

$$\begin{bmatrix} \textbf{throwing} & \\ \text{THROWER:} & \begin{bmatrix} \textbf{index} & \\ \text{PERSON:} & \textbf{third} \\ \text{NUMBER:} & \textbf{[0]singular} \\ \text{GENDER:} & \textbf{masculine} \end{bmatrix} \\ \text{THROWN:} & \begin{bmatrix} \textbf{index} & \\ \text{PERSON:} & \textbf{third} \\ \text{NUMBER:} & \textbf{[0]} \\ \text{GENDER:} & \textbf{neuter} \end{bmatrix} \end{bmatrix}$$

Figure 2.6: Structure sharing in an AVM representation of a TFS.

structure of type **index**, which bear the feature NUMBER having as value the same type **singular**. Structure sharing avoids duplicating such information, by keeping a single instance of the node labeled with the type **singular**. Figure 2.5 shows this feature structure with the node **singular** shared, while in Figure 2.6, the AVM representation is shown (where the sharing is indicated by the tag [**0**]).

Structure sharing could lead to cyclic structures (cycles in the graph representing a feature structure), when a node has a substructure that incorporates that node. An example of such a cycle can be found in the feature structure representation for "*every book*" (adapted from [Pollard and Sag, 1994, p.50, p.361]):

$$
\left[\begin{array}{ll}
\text{SPR:} & \left[\begin{array}{ll}
\text{PHON: } \textbf{every} & \\
\text{SYNSEM:} & \left[\begin{array}{l}
[\mathbf{2}] \\
\text{LOC: } \left[\text{CAT: } \left[\text{HEAD: } \left[\begin{array}{l} \textbf{det} \\ \text{SPEC: } [\mathbf{1}] \end{array}\right]\right]\right]
\end{array}\right]
\end{array}\right] \\
\text{HEAD:} & \left[\begin{array}{ll}
\text{PHON: } \textbf{book} & \\
\text{SYNSEM:} & \left[\begin{array}{l}
[\mathbf{1}] \\
\text{LOC: } \left[\text{CAT: } \left[\text{SPR: } [\mathbf{2}]\right]\right]
\end{array}\right]
\end{array}\right]
\end{array}\right].
$$

2.5 Typed Feature Structure Cuts

In certain applications (as it will be the case for the indexing methods introduced in Chapter 6), it is useful to treat a subset of the nodes in a TFS in a similar manner as TFSs. In this thesis, the concept of *TFS cuts* is introduced in order to provide a TFS-like treatment to subset of nodes.

Definition 2.19. *A cut of a typed feature structure* $F = \langle Q, \overline{q}, \theta, \delta \rangle$ *is a tuple* $F^{\text{✂}} = \langle Q', \overline{Q'}, \theta, \delta \rangle$ *where:*

- $Q' \subseteq Q$ *(*$Q' \neq \emptyset$*), a finite set of nodes,*
- $\overline{Q'} = \{\langle x, \Pi(x) \rangle | x \in Q'$ *and* $\exists y \in Q \setminus Q', \exists f.\delta(f, y) = x\}$*, where* $\Pi(x) = \{\pi | \delta(\pi, \overline{q}) = x\}$*, the set of pseudo-roots of* $F^{\text{✂}}$*.*

The definition for the unification of typed feature structures can now be extended to allow for the unification of a typed feature structure with a cut of a typed feature structure. An example of a unification between a TFS and a cut is presented in Figure 2.7.

Definition 2.20. *Suppose* F, G *are typed feature structures such that* $F \sim \langle Q_F, \overline{q_F}, \theta, \delta\rangle$, $G \sim \langle Q'_G, \overline{q_G}', \theta', \delta'\rangle$, $Q_F \cap Q_G = \emptyset$, *and* $F^{\text{✂}} \sim \langle Q'_F, \overline{Q'_F}, \theta, \delta\rangle$ *is a cut of* F. *The equivalence relation* $\bowtie$ *on* $Q'_F \cup Q_G$ *is defined as the least equivalence relation such that:*

1) *if* $\langle q_F, \Pi(q_F)\rangle \in \overline{Q'_F}$ *and* $q_G \in Q_G$, *then* $q_F \bowtie q_G$ *iff* $\exists \pi \in \Pi(q_F)$ *such that* $\delta'(\pi, \overline{q_G}) = q_G$,

2) *if* $\delta(f, q_F)\downarrow$ *and* $\delta'(f, q_G)\downarrow$, *and* $q_F \bowtie q_G$, *then* $\delta(f, q_F) \bowtie \delta'(f, q_G)$.

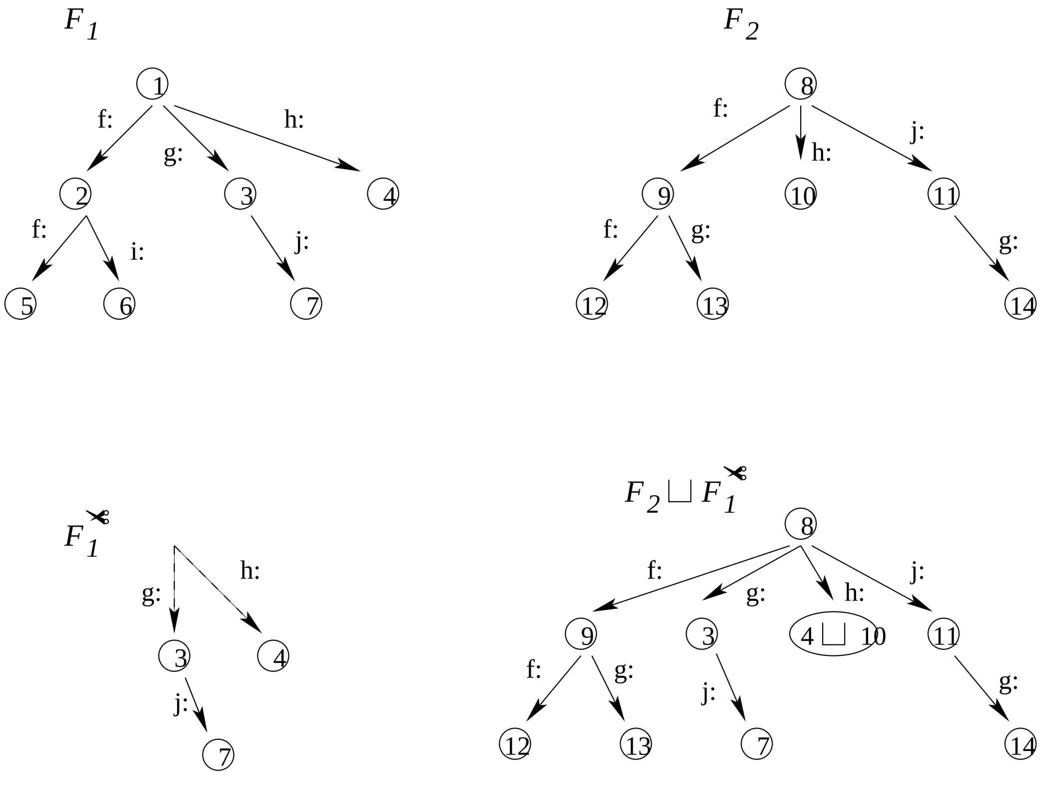


Figure 2.7: The unification between a TFS and a cut TFS.

Proposition 2.1. *Suppose F, G are typed feature structures such that $F \sim \langle Q_F, \overline{q_F}, \theta, \delta\rangle$, $G \sim \langle Q'_G, \overline{q_G}', \theta', \delta'\rangle$, $Q_F \cap Q_G = \emptyset$, $F^{\rightthreetimes} \sim \langle Q'_F, \overline{Q'_F}, \theta, \delta\rangle$ is a cut of F, $\bowtie$ is the equivalence relation between F and G (from the unification definition), and $\blacktriangleright\!\blacktriangleleft$ is the equivalence relation between $F^{\rightthreetimes}$ and G (from Definition 2.20). If $x \in Q'_F, y \in Q_G$, then $x \blacktriangleright\!\blacktriangleleft y \Leftrightarrow x \bowtie y$.*

Proof. (By induction on least distance from a pseudo-root to a node $x \in Q'_F$)

Part 1: $x \blacktriangleright\!\blacktriangleleft y \Rightarrow x \bowtie y$.

Base case: $\langle x, \Pi\rangle \in \overline{Q'_F}$.

Since $x \blacktriangleright\!\blacktriangleleft y$, $\exists \pi \in \Pi$ such that $\delta'(\pi, \overline{q_G}) = y$. By definition of $\rightthreetimes$, $\delta(\pi, \overline{q_F}) = x$; and by definition of $\bowtie$, $\overline{q_F} \bowtie \overline{q_G}$. Thus, according to the definition extending δ to paths, and according to the Condition 2 from the definition of $\bowtie$: $\delta(\pi, \overline{q_F}) \bowtie \delta'(\pi, \overline{q_G})$. Therefore, $x \bowtie y$.

Induction: Suppose $x' \in Q'_F$ and $y' \in Q_G$ such that $x' \blacktriangleright\!\blacktriangleleft y'$ and $x' \bowtie y'$. According to Condition 2 in Definition 2.20, if $\exists x = \delta(f, x')$ and $\exists y = \delta(f, y')$, then $x \blacktriangleright\!\blacktriangleleft y$. However, $x' \bowtie y'$, and according to the definition of $\bowtie$, if $\delta(f, x') \downarrow$ and $\delta(f, y') \downarrow$, then $\delta(f, x') \bowtie \delta(f, y')$. Therefore, $x \bowtie y$.

Part 2: $x \blacktriangleright\!\blacktriangleleft y \Leftarrow x \bowtie y$. Similarly to $x \blacktriangleright\!\blacktriangleleft y \Rightarrow x \bowtie y$.

$\square$

Chapter 3

Unification-Based Grammars

Typed Feature Structures can be employed in various areas of Computational Linguistics, or Natural Language Processing, such as parsing, question-answering, information retrieval, etc. Since the focus of this thesis is on parsing with Typed Feature Structures, this chapter begins with a short overview of HPSG, a representative unification-based grammar formalism using typed feature structures. HPSG is also the formalism of the grammar used in the experimental evaluation in Chapter 7 (although the indexing methods introduced in this thesis are not limited to parsing with HPSG). After introducing the HPSG formalism, this chapter concludes with an extensive presentation of various implementational aspects of Typed Feature Structure Grammars, presentation that also introduces a new classification of the instances of variables used in TFSGs.

Unification-based grammars (UBGs), as suggested by their name, are grammar formalisms where unification is the only information-combining operation [Shieber, 1986]. More specifically, entities are represented by feature structures, and UBGs allow information carried by such entities to be combined only through unification.

Some particular properties differentiate UBGs from simpler formalisms, such as CFGs. According to [Shieber, 1986], UBGs are:

Inductive: the associations between entities and strings can be defined recursively.

Declarative: the grammar formalism specifies what associations between entities and strings are allowed, without necessarily specifying how the associations are built.

Complex-feature-based: entities are expressed as feature-value associations in a well-defined and structured domain.

In the following section, the HPSG formalism is introduced. It is not the purpose of this work to extensively present unification-based formalisms; wide-coverage surveys of UBGs can be found in [Shieber, 1986] and [Brown and Miller, 1996].

This chapter also introduces a new classification of the instances of variables used in descriptions, based on what type of structure sharing they create. The use of variables to share structure is a powerful characteristic of TFSGs.

3.1 Head-Driven Phrase Structure Grammars

Head-Driven Phrase Structure Grammars (HPSGs) were first introduced in [Pollard and Sag, 1987], while the complete theoretical framework was published later in [Pollard and Sag, 1994]. The HPSG formalism was designed as a refinement of Generalized Phrase Structure Grammar, and similarly to GPSG, describes a linguistic theory [Shieber, 1986].

3.1.1 Principles of HPSG

There are several reasons for the continually increasing popularity of HPSG. Cooper [1996] notes that even if HPSG – as linguistic formalism – is a combination of various theories, it offers a rigorous framework for grammar development through its foundation on unification of feature structures. Shieber [1986] mentions the support for subcategorization and semantic representation as important qualities of HPSGs. Furthermore, Uszkoreit [1996] lists key components of the HPSG formalism that explain the impact it had on grammar development, such as:

- its foundation on typed feature logic,
- using the same formalism for universal and particular generalizations,
- allowing for language-specific analyses, that are less natural described in other formalisms (e.g. parasitic gaps, clitics in French, phrase structure in German, etc.)

The *Head-Driven* component of the HPSG name reveals one of the underlying principles of this formalism. The most important element of a phrase is its lexical head [Pollard and Sag, 1994]. The lexical head incorporates both syntactic information (such as part of speech and dependency relations with other constituents) and semantic information. Lexical entities, in general, are information-rich structures, with feature values from the type signature (an inheritance hierarchy, as introduced in Chapter 2). This lexical-centered organization of HPSG eliminate redundancies, therefore the amount and complexity of phrasal rules is greatly reduced [Cooper, 1996].

3.1.2 The HPSG Formalism

Formally, HPSG is organized as a set of grammar principles, rules, and lexical descriptions [Cooper, 1996]. Each constituent is described by a typed feature structure, as in the examples presented in Chapter 2. HPSG is a refinement of the GPSG formalism, and from a linguistic perspective it shares similar principles [Meurers, 2002]: lexical entries, lexical rules, Immediate Dominance (ID) rules (specifying a dominance relation between categories, without imposing an order of categories), Linear Precedence (LP) rules (defining and order relation between some daughters in a rule; these, together with ID rules, are the grammar rules), and grammatical principles. To an extent, a HPSG can be viewed as a feature structure that must satisfy a description of the form:

$$P_1 \wedge \cdots \wedge P_k \wedge (R_1 \vee \cdots \vee R_m \vee L_1 \vee \cdots \vee L_n)$$

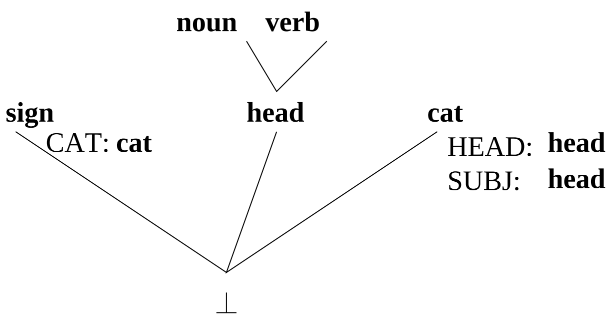
where $P_1 \ldots P_k$ are grammatical principles, $R_1 \ldots R_m$ are grammar rules, and $L_1 \ldots L_n$ are lexical entries.

One of the major differences between HPSG and formalisms such as CFG is the underlying mathematical structure [Pollard, 1997]: the CFG formalism is based on sequences of phrase-markers (trees where nodes are labeled with categories), while HPSG is based on feature structures (which represent linguistic entities in graphs where arcs are labeled with feature names and nodes with types of linguistic objects). The relation between entities in HPSG is formulated as well-formedness constraints on typed feature structures. Another major difference between HPSG and CFG-like formalisms is that HPSGs are considered to be nonderivational, meaning there are no transformations or destructive operations to derive one structure from another[Pollard, 1997].

Figure 3.1 presents a simple example [Matheson, 1997] of a HPSG. The type hierarchy contains three basic types (`sign`, `cat`, and `head`), where `head` has two subtypes (`noun` and `verb`). Appropriateness is specified through declarations like `CAT:cat` together with the type `sign`. It is said that `sign` *introduces* the feature `CAT`, meaning `sign` is the least type for which the feature `CAT` is appropriate. This declaration also specifies that possible values for `CAT` have to be of type `cat`[1] or one of its subtypes.

Lexical entries are specified using the descriptions presented in Section 2.3.3. For example, the lexical head of "snored" is `verb`, and the subject of "snored" is a `noun`. Although the description for "Mary" might

[1] There is a distinction between upper and lower case. It is common to HPSG to have features and feature values bearing the same name.



$$mary \longrightarrow CAT : HEAD : noun$$

$$snored \longrightarrow CAT : (HEAD : verb \wedge SUBJ : noun)$$

$$(CAT : HEAD : verb) \Longrightarrow$$

$$(sign \wedge CAT : HEAD : HeadVar), (sign \wedge CAT : SUBJ : HeadVar)$$

Figure 3.1: A Simple HPSG

suggest that it only specifies a `noun`, total well-typedness guarantees the presence of all appropriate features:

$$mary \longrightarrow \begin{bmatrix} \textbf{sign} & \\ \text{CAT:} & \begin{bmatrix} \textbf{cat} \\ \text{HEAD: } \textbf{noun} \\ \text{SUBJ: } \textbf{head} \end{bmatrix} \end{bmatrix}.$$

This simple grammar also contains a rule implementing the subject-head schema. Similar to the lexical entry for "Mary", the descriptions in rules are specified only at features having values different than ones inferable from appropriateness. The rule states that the mother has a `verb` as its head (meaning the head of the phrase is a verb), while the first daughter (the subject) has the same value for its feature `head` as the second daughter (the head) has for its `subj` feature. This is achieved through the use of the variable `HeadVar`, which creates structure sharing between the most general satisfiers of the two daughters.

HPSG is a complex formalism, under constant improvement through various extensions. It is outside the scope of this work to present an in-depth overview of HPSG. More formal details can be found in [Pollard and Sag, 1987; 1994] or [Cooper, 1996]. A concise introduction can be found in [Pollard, 1997]. Information about current research trends and HPSG developments is maintained on several HPSG-related web servers: http://www.sfs.uni-tuebingen.de/hpsg/, http://hpsg.stanford.edu/, http://www.ling.ohio-state.edu/research/hpsg/, etc.

3.2 Implementation Aspects

This section focuses on two aspects related to the implementation of UBGs (exemplified through HPSGs). The first aspect is the transition from a formal specification of the grammar rules to a computer-usable form. The second aspect is a narrowing of the first one to the specific details of representing TFSs.

3.2.1 Compiling Grammar Rules

As mentioned in Section 2.3.3, the constraints reflected in a TFS can be expressed through descriptions. Without getting into explicit details about various description languages, this section presents the rationale behind the transformation of grammar rules into a representation usable in a programming environment (specifically, in Prolog). For this, the Attribute Logic Engine (ALE) will be used as the example parsing system.

ALE [Carpenter and Penn, 2001] is a phrase structure parsing system, supporting various formalisms, such as HPSG. Its grammar handling mechanism is built on foundations of the Prolog built-in DCG system, with the important difference of using descriptions of TFSs for representing categories, instead of Prolog terms.

There are two main components in the grammar handling mechanism: the lexicon and the grammar rules. The lexicon consists of lexical entries and lexical rules. Lexical rules are used to express the redundancies among lexical entries.

Of interest to the work presented in this thesis are the grammar rules. An example of a phrase rule in ALE is given in Figure 3.2.

Using the descriptions presented in Figure 3.2 in an implementation would not be practical (not only for efficiency reasons, but also because types can be promoted, and Prolog variables, once instantiated, cannot be changed). Therefore, the grammar is first compiled into an internal, efficient, representation. The choice for the internal representation of each category in the grammar depends on the programming language that is chosen for the implementation. For the particular case of Prolog, the next section presents several encodings of TFSs.

Another reason for the off-line compilation of the grammar is the possibility of performing several optimizations. As it will be shown later in this thesis, an analysis of grammar rules carried out at compile-time results

```
s_np_vp rule
(syn:s,
 sem:(VPSem,
      agent:NPSem))
===>
cat>
  (syn:np,
   agr:Agr,
   sem:NPSem),
cat>
  (syn:vp,
   agr:Agr,
   sem:VPSem).
```

Figure 3.2: A phrase rule in ALE. A line of the form `syn:vp` represents the description of a feature named `syn` (syntactic category) with the value (type restriction) `vp`. The above rule states that the syntactic category `s` can be combined from `np` and `vp` categories if their values for the feature `agr` are the same. The semantics of `s` is the semantics of the verb phrase, while the role of agent is served by the semantics of the noun phrase.

in a better indexing scheme, leading to faster parsing times.

3.2.2 Variables in TFS Representations

A particular point of interest with regard to the representation of grammar rules is the use of variables. The unification and instantiation mechanisms in Prolog allow the use of variables to represent structure shared between daughters or between a daughter and the rule's mother. The example from Figure 3.2 is significant from this point of view: the features `agr` in the two daughters share the same structure, indicated by the variable `Agr`. Similarly, semantic information is shared between the mother and each of the two daughters through the variables `VPSem` and `NPSem`.

The use of variables to share structure is a powerful characteristic of TFSGs. This thesis introduces a new clasification of the instances of variables, based on what type of structure sharing they create: *internal variables*, *active external variables*, and *inactive external variables*. Over the next chapters, the importance of these variables for indexing will be revealed.

Internal variables: the instances of these variables represent internal structure sharing. An example of such internal sharing can be found in the AVM from Figure 2.6 on page 21. The occurrences of the instances of such variables are limited to a single category in the grammar.

External Variables: are instances of variables such as `Agr`, `NPSem`, and `VPSem` from Figure 3.2. They are used to share structure across categories (daughters). These instances can occur in the entire grammar rule. It is possible for a variable instance to be used for structure sharing both inside a category (as an internal variable) and across categories; in this case, it is considered an external variable. For a specific category, two kinds of external variables can be distinguished, depending on their sharing with other categories and the parsing control strategy:

Active External Variables: are instances of variables that are shared between the description of a category D and descriptions of daughters in the same rule with D that are visited before D when the rule is extended. For example, if rules are extended from left to right, then the Active External Variables for a daughter D_i are those variable instances shared between D_i and all daughters D_j with $j < i$ (daughters to the left of D_i). Since the evaluation is left to right, a daughter sitting to the right of D_i sharing a variable with D_i cannot have that variable instantiated before D_i (i.e., external variables get instantiated in D_i because daughters at its left were unified with edges in the chart[2]). For a mother, all its External Variables are Active External Variables.

Inactive External Variables: are the External Variable instances that are not active.

For the rest of this thesis, unless otherwise specified, a daughter's External Variables will refer to its Active External Variables.

A particular variable can be active for a daughter D_i, but inactive for a daughter D_j in the same rule. In Figure 3.2, `Agr` is an Active External

[2]A discussion about TFSG parsing is presented in Section 4.1.

Variable in the second daughter, but it is an Inactive External Variable in the first daughter.

When a description is mapped to a feature structure with MGSat, External Variables correspond to structure sharing between feature structures ("external sharing"), as exemplified in Figure 3.3.

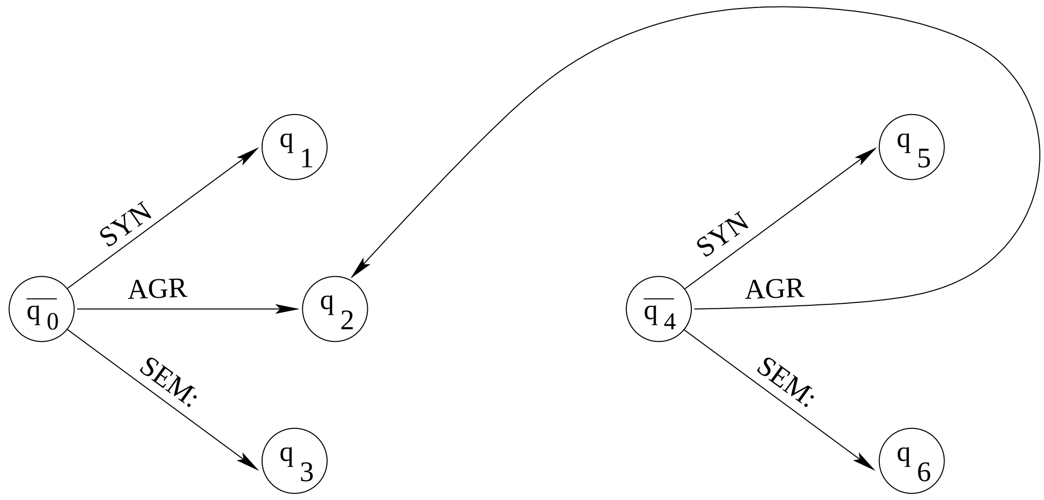


Figure 3.3: Structure sharing between two typed feature structures ("external sharing").

It should be mentioned that by external sharing, a description's MGSat can "grow" nodes. For the example presented in Figure 3.2 and Figure 3.3, if the first daughter is unified with an edge

$$\begin{bmatrix} \text{SYN: } \textbf{np} \\ \text{AGR: } \begin{bmatrix} \text{PERS: } \textbf{third} \\ \text{NUMBER: } \textbf{sing} \end{bmatrix} \end{bmatrix},$$

then the two extra nodes connected to node q_2 in this daughter will also be shared by the second daughter, as shown in Figure 3.4.

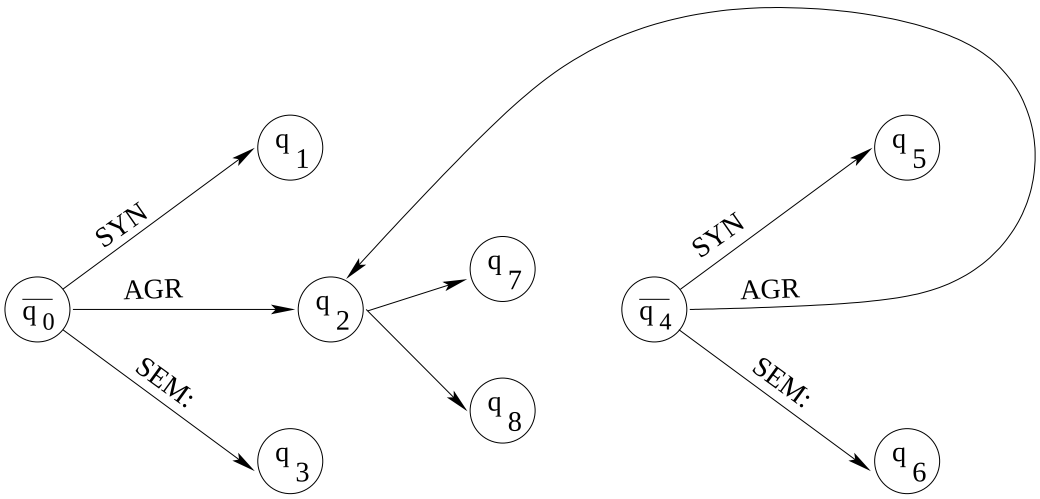


Figure 3.4: Structure sharing between two typed feature structures, after unification.

Structure sharing (especially external sharing) is an important aspect of typed feature structures. Its relevance to parsing will be detailed in Chapter 6; but before closing this section, some extra notation is provided:

Definition 3.1. *If $D_1, \ldots, D_n$ are daughter descriptions in a rule, then $Ext(MGSat(D_i))$ is the set of nodes shared between $MGSat(D_i)$ and the daughters to its left in the same rule: $Ext(MGSat(D_i)) = \bigcap_{1\leq j<i} Q_j$, where Q_j is $MGSat(D_j)$'s set of nodes.*

For a mother description M, $Ext(MGSat(M))$ is the set of nodes $q \in Q_M$ shared with any daughter in the same rule: $Ext(MGSat(M)) = \bigcap_{1\leq j\leq n} Q_j$, where n is the number of daughters in the rule, Q_j is $MGSat(D_j)$'s set of nodes, and Q_M is $MGSat(M)$'s set of nodes.

For the result of the unification between M and D, $Ext(MGSat(M) \sqcup$

$MGSat(D)) = \{[x]_{\bowtie} \in Q_{M \sqcup D} | \exists y \in [x]_{\bowtie}$ *such that* $y \in Ext(MGSat(M)) \cup Ext(MGSat(D))\}$.

It should be mentioned that $Ext(MGSat(M)) \cup Ext(MGSat(D))$ needs to be defined since daughters are unified during parsing with mothers inserted as edges in the chart. It should also be noted that Ext is defined for MGSats of daughter and mother descriptions. For the rest of the thesis, when not specified otherwise, a daughter D (and a mother M) will denote the most general satisfier of a daughter description (and of a mother description).

3.2.3 TFS Grammar Rules

The previous section presented instances of variables in logical descriptions as the source for information sharing. In TFSG parser implementations however, the structure sharing is observed at the node level. This creates difficulties in formalizing the concept of structure sharing across most general satisfiers of daughters in the same rule.

Wintner [1997] proposes a formal definition of phrase structure rules using feature structures. In this formalization, a rule is regarded as a feature structure with multiple roots, where each root corresponds to the root of the rule's mother and daughters. A version of the multi-rooted feature structures, adapted to the domain of this thesis, is given here.

Before formally introducing the phrase rule definition, several supporting definitions must be given.

Definition 3.2. *A* multi-rooted typed feature structure (MRS) *over Type and Feat is a tuple* $R = \langle Q, \overline{Q}, \theta, \delta \rangle$ *where:*

- *Q is a finite set of nodes,*
- *$\overline{Q} \in Q$ is an ordered sequence of roots (possibly containing duplicates),*
- *$\theta : Q \to Type$ is a total node typing function,*
- *$\delta : Feat \times Q \to Q$ is a partial feature value function*

such that for all $q \in Q$, there exists $\overline{q} \in \overline{Q}$ and there exists a finite sequence of features $f_1, \ldots, f_n \in Feat$ that connects $\overline{q}$ to q with δ: $q = \delta(f_n, \ldots \delta(f_2, \delta(f_1, \overline{q})))$. The length *of a MRS is the number of its roots, $|\overline{Q}|$.*

A MRS can be seen as an ordered sequence of feature structures $R = \langle F_1, \ldots, F_n \rangle$ with shared nodes. If $\overline{q_i}$ is a root in $\overline{Q}$, then F_i is the feature structure induced by the i-th root of R. F_i contains all the nodes in Q that are reachable from $\overline{q_i}$.

Definition 3.3. *If $R = \langle Q, \overline{Q}, \theta, \delta \rangle$ is a MRS, then its* induced feature structures *are $F_1, \ldots, F_n$ such that $\forall i \in (1, \ldots, n)$, $F_i = \langle Q_i, \overline{q_i}, \theta, \delta \rangle$ where:*

- *$\overline{q_i}$ is the root node at position i in $\overline{Q}$,*
- *Q_i is the largest set of nodes $\{q \in Q\}$ such that $\exists \pi . \delta(\pi, \overline{q_i}) = q$.*

The unification of two MRSs can be defined as follows:

Definition 3.4. *Suppose R and R' are multi-rooted feature structures of length n such that $R = \langle Q, \overline{Q}, \theta, \delta \rangle$ and $R' = \langle Q', \overline{Q}', \theta', \delta' \rangle$. The equivalence relation $\bowtie$ on $Q \cup Q'$ is defined as:*

- *$\overline{q_i} \bowtie \overline{q_i}', \forall i \in (1, \ldots, n)$,*

- *if* $\delta(f,q)$ *and* $\delta'(f,q')$ *are defined, and* $q \bowtie q'$*, then* $\delta(f,q) \bowtie \delta'(f,q')$.

The unification of R *and* R' *is then defined as:*

$$R \sqcup R' = \langle (Q \cup Q')/_{\bowtie}, \overline{Q}/_{\bowtie}, \theta^{\bowtie}, \delta^{\bowtie} \rangle$$

where

$$\theta^{\bowtie}([q]_{\bowtie}) = \bigsqcup \{(\theta \cup \theta')(q') | q' \bowtie q\}$$

and

$$\delta^{\bowtie}(f, [q]_{\bowtie}) = \begin{cases} [(\delta \cup \delta')(f,q)]_{\bowtie}, & \text{if } (\delta \cup \delta')(f,q) \text{ is defined} \\ \text{undefined}, & \text{otherwise} \end{cases}$$

if all the joins in the definition of $\theta^{\bowtie}$ *exist.* $R \sqcup R'$ *is undefined otherwise.*

A phrase structure rule can now be seen as a MRS, where each constituent is a TFS in the MRS.

Definition 3.5. *A phrase rule is a multi-rooted feature structure* $R = \langle F_1, \ldots, F_n \rangle$ *of length* $n > 0$ *with a distinguished last element.* F_n *is the mother, while* $F_1, \ldots, F_{n-1}$ *are the daughters:* $F_1, \ldots, F_{n-1} \Rightarrow F_n$.

Figure 3.5 presents Wintner's example of a TFS rule seen as a MRS [Wintner, 1997]. The rule is written as $F_1, \ldots, F_{n-1} \Rightarrow F_n$ (being equivalent to a derivational rule: $F_n \rightarrow F_1, \ldots, F_{n-1}$.)

3.2.4 Encoding of TFSs

The indexing methods proposed in Chapters 4 and 6 are designed to be implementation-independent. However, for practical reasons, and to relate

$$\begin{bmatrix} \textbf{phrase} & \\ \text{SYN:} & \textbf{n} \\ \text{HEAD:} & [1]\begin{bmatrix} \textbf{head} & \\ \text{AGR:} & [3]\textbf{agr} \end{bmatrix} \\ \text{CASE:} & \textbf{nom} \end{bmatrix} \begin{bmatrix} \textbf{phrase} & \\ \text{SYN:} & \textbf{v} \\ \text{HEAD:} & [2]\begin{bmatrix} \textbf{head} & \\ \text{AGR:} & [3]\textbf{agr} \end{bmatrix} \\ \text{SUBCAT:} & \textbf{elist} \end{bmatrix} \Rightarrow \begin{bmatrix} \textbf{phrase} & \\ \text{SYN:} & \textbf{s} \\ \text{SUBJ:} & [1] \\ \text{HEAD:} & [2] \end{bmatrix}$$

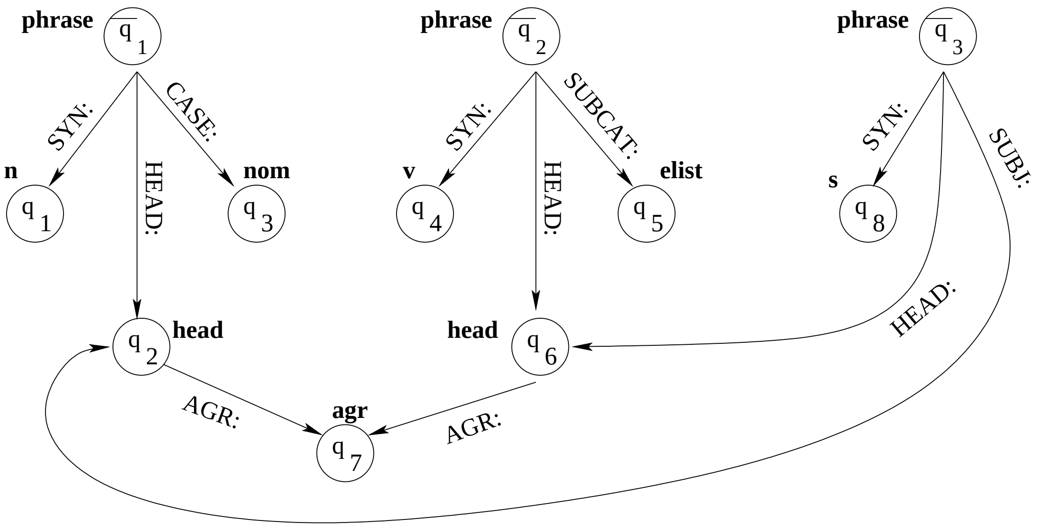


Figure 3.5: A phrase rule seen as a MRS.

to existing systems, the experimental evaluation is carried out using a Prolog implementation, namely ALE[Carpenter and Penn, 2001]. For this reason[3], this section outlines existing methods for encoding (representing) typed feature structures in Prolog.

In his work, Mellish [1988; 1991] formalized the definition of term

[3]The reason ALE and other similar systems are implemented in Prolog is thoroughly explained in [Penn, 2000] and [Penn and Munteanu, 2003].

encoding as a function from a type hierarchy to a set of Prolog terms which is [Penn, 1999a]:

- injective (one term encodes no more than one type),
- homomorphic with respect to unification (the encoding preserves the unifications in the type hierarchy),
- zero-preserving (the encoding preserves the failure of unifications in the type hierarchy).

Using Colmerauers's method [Colmerauer, 1982], Mellish [Mellish, 1992] showed that all hierarchies can be encoded in a flat-term encoding, using no more than $n+1$ arguments (where n is the number of types in the hierarchy.)

TFSG signatures contain not only a type hierarchy, but also a collection of features. A straightforward encoding would need a number of argument positions equal to the number of features. This is not desirable, and Penn [1999a] proposed a method for reducing the arity of terms in a flat encoding. For this, a graph is defined as an undirected graph where vertices correspond to features and each edge connects two features that are appropriate for a common maximally specific type. The least number n for which the feature graph is n-colourable is the least arity necessary for the encoded terms. This encoding will be used in the preliminary experimental evaluation presented in Chapter 7 (specific details will be given in Chapter 7.)

An optimized approach is taken in [Penn, 1999b], where features are encoded based on their feature-graph colouring, but the argument position for each feature is statistically chosen. Features that are considered to be more likely to cause unification failures are placed in a lower argument position.

This optimized encoding forces an earlier failure when unifying two typed feature structures, therefore saving parsing time.

Chapter 4

Indexing for TFSG Parsing

This chapter introduces a general indexing method that can be applied to any chart-based parser. Without the loss of generality, and for preserving consistency, the chart parsing algorithm used for illustration here is an efficient algorithm based on EFD (Empty-First-Daughter) closure [Penn and Munteanu, 2003], implemented in Prolog. Before giving a detailed presentation of the proposed indexing method in Section 4.3, two additional concepts are introduced: a connection between parsing and typed feature structures and an indexing timeline that can be applied to any chart-based parser. The chapter concludes with an overview of related approaches to indexing/filtering TFSGs, accompanied by a discussion about advantages and disadvantages of the existing and proposed methods.

4.1 Chart Parsing with Typed Feature Structure Grammars

In this section, several problems related to parsing grammars based on typed feature structures are outlined. In the first subsection, the most well-known obstacles to efficient parsing with TFSGs are introduced, together with details about parsing with TFSGs. The second subsection presents a textbook parsing algorithm suitable for TFSGs, while the third section gives an overview of an efficient parsing algorithm used as baseline for the experimental evaluation in Chapter 7. Finally, a formal definition of the rule completion process in TFSG chart parsing is introduced.

4.1.1 Parsing with Typed Feature Structures

Parsing with Typed Feature Structures is an issue that has raised many concerns among researchers in Computational Linguistics. When TFSs are used in formalisms such as HPSG, the parsing problem is generated by two aspects [van Noord, 1997]: the complex and large structure of the lexical entries, and the small number of grammar rules. The following paragraphs present a motivation for the choice of bottom-up, all-paths, parsing algorithms as suitable for TFSGs.

While the size of lexical entries in TFSGs is a concern from the perspective of increased unification times, the grammar rules are related directly to the parsing algorithm. As mentioned in [Oepen and Carroll, 2000], strictly top-down parsing algorithms are not suitable for TFSGs (a complete comparison between bottom-up and top-down parsers can be found in [Jurafsky and Martin, 2000]).

The motivation for using bottom-up parsers for TFSGs does not lie solely in the structure of the grammar. As van Noord [1997] observes, a top-down parser may encounter termination problems. Using enhanced top-down parsing techniques, such as restricted top-down predictions, avoids the termination issues, but generally results in degraded performance.

Finally, bottom-up parsing is suitable for TFSGs due to the lexicalist nature of such formalisms [van Noord, 1997]. Since bottom-up parses "never suggest trees that are not at least locally grounded in the actual input" [Jurafsky and Martin, 2000], this class of parsing algorithms seems more adequate for a grammar formalism where the syntactic and semantic information is heavily concentrated in the lexical entries.

Another important aspect when parsing TFSGs (and UBGs in general) is the need for an all-paths parser. The reason behind this need is given in [Penn and Munteanu, 2003]:

> While there have been great advances in probabilistic parsing methods in the last five years, which find one or a few most probable parses for a string relative to some grammar, all-paths parsing is still widely used in grammar development, and as a means of verifying the accuracy of syntactically more precise grammars, given a corpus or test suite.

4.1.2 Bottom-Up Parsing

Many parsing techniques are available, some of them being designed long before TFSGs were developed. As mentioned in the previous section, one of the most suitable technique for TFSGs is bottom-up parsing. Most natural language processing or computational linguistics textbooks ([Allen, 1994], [Jurafsky and Martin, 2000], [Gazdar and Mellish, 1989], [Pereira and Shieber, 1987]) include detailed presentations of bottom-up parsing. Therefore, just some notational details are required here.

Bottom-up parsing can traverse the input string from right to left or from left to right. When not specified, it will be assumed here that the traversal is right to left (this is the approach taken in the EFD parsing method, described in the next section.) Rules also can be processed in both directions; the default assumption will be that the processing is left to right (as in the EFD algorithm). Several books (such as [Allen, 1994]) use the term **active arc** to denote a rule in the course of being completed; in this thesis, the term **active edge** will be used. Similarly, constituents in the chart are named **edges** (or **passive edges**) instead of (**passive**) **arcs**. The left-hand side of a rule is the **mother** of the rule, while categories on the right-hand side are **daughters**. The process of traversing a rule and matching its daughters with edges in the chart is called the **completion** of the rule.

4.1.3 EFD Parsing

The Empty-First-Daughter (EFD) parsing method is employed to support the indexing techniques introduced in this thesis. It is also used as the baseline parser in comparison with the improved, indexed, parser. This section outlines the principles of this algorithm; a detailed presentation can be found in [Penn, 1999c] and [Penn and Munteanu, 2003]

An important issue in bottom-up parsing, when implemented in Prolog, is the amount of copying. In standard Prolog parsers, each time an edge is matched with a daughter (regardless of the success of the matching operation), the edge is copied from the assertional database onto the heap. This copying can significantly slow down the parsing process when categories are large (such as typed feature structures).

A first approach to solving the copying problem was Carpenter's algorithm, which is used in ALE [Carpenter and Penn, 2001]. The algorithm traverses the input sentence breadth-first and from right to left. The grammar rules are traversed depth-first, from left to right, in a failure-driven loop. As a result, Carpenter's algorithm does not require active edges, since only one rule at a time is in the process of being completed. This is in contrast to a breadth-first traversal of rules, where a record of all rules that have matched partially with edges in the chart (but are not completed yet) must be kept.

The EFD parsing algorithm brings further improvements to Carpenter's algorithm, by limiting the number of copying operations to a maximum of two per edge. Apart from the benefit of reducing copying, the EFD parsing algorithm offers a better support for indexing, especially for Prolog implementations. The chart is not stored in the assertional database, but on the heap. Thus, indexing can be applied without copying edges.

The EFD algorithm is based on the assumption that the grammar is *EFD-closed*: all rules in the grammar have at least one daughter, and the leftmost daughter of each rule is blocked from deriving an empty category. The algorithm for transforming a phrase-structure grammar into an EFD-closed grammar is presented in [Penn and Munteanu, 2003]. During parsing, after an edge is added to the chart, a failure driven-loop visits the rules in the grammar and selects the ones for which the first (the leftmost) daughter unifies with the newly created edge. The chosen rule is then extended from left to right, starting with its second daughter, by matching the daughters with edges in the chart and empty categories.

4.1.4 Parsing as Unification of MRSs

The benefit of regarding phrase rules as MRSs is reflected in the simplicity of defining the basic operation of parsing: rule completion. If a rule has n daughters, and the rule is represented as an MRS $R = \langle D_1, \ldots, D_n, M\rangle$ (where $D_1, \ldots, D_n$ are MGSats of the rule's daughters and M is the MGSat of the mother), then completing the rule is achieved by obtaining the sequence $R^{\widehat{1}}, \ldots, R^{\widehat{n}}$, where each intermediate MRS $R^{\widehat{i}}$ represents the rule (the active edge) after D_i is unified with an edge in the chart. The following paragraphs introduce a formal definition of the rule completion process.

Definition 4.1. *If F is a typed feature structure $\langle Q, \overline{q}, \theta, \delta\rangle$, then $\langle F, i, n\rangle$,* the expansion of F to position i in a MRS of length n, *is $\langle Q', \overline{Q}, \theta, \delta\rangle$, where:*

- $Q' = Q \cup \overline{Q'}$, *where* $\overline{Q'} = \{\overline{q'_j} | 1 \le j \le n, j \ne i, \theta(\overline{q'_j}) = \bot\}$,
- $\overline{Q} = \{\overline{q_j} | 1 \le j \le n\}$ *is an ordered set such that* $\overline{q_j} = \begin{cases} \overline{q}, & \text{if } j = i \\ \overline{q'_j}, & \text{otherwise} \end{cases}$.

Definition 4.2. *For a rule represented by a MRS $R = \langle D_1, \ldots D_n, M\rangle$, its first active edge (after D_1 is unified with an edge E_1 from the chart) is an MRS $R^{\widehat{1}} = R \sqcup \langle E_1, 1, n\rangle$. Consequently, its i-th active edge (after D_i, $1 < i \le n$, is unified with an edge E_i from the chart) is an MRS $R^{\widehat{i}} = R^{\widehat{i-1}} \sqcup \langle E_i, i, n\rangle$.*

If $R = \langle D_1, \ldots D_n, M \rangle$ is an MRS representing a rule, and $R^{\widehat{i}}$ is its i-th active edge, then $R^{\widehat{i}}$ can be written as $\langle {D_1}^{\widehat{i}}, \ldots {D_n}^{\widehat{i}}, M^{\widehat{i}} \rangle$. ${D_1}^{\widehat{i}}, \ldots {D_n}^{\widehat{i}}$ represent R's daughters (and $M^{\widehat{i}}$ its mother) after the first i daughters are unified with edges in the chart. As underlined by Wintner [1997], the TFSs induced by a MRS are not (necessarily) independent, due to the nodes shared between them. The following definition introduces the relation (represented by the mapping operator $\mapsto$) between nodes in R and $R^{\widehat{i}}$:

Definition 4.3. *If $R^{\widehat{0}} = R = \langle Q, \overline{Q}, \theta, \delta \rangle$ is a MRS of length n representing a rule and $R^{\widehat{i}} = \langle Q^{\widehat{i}}, \overline{Q}^{\widehat{i}}, \theta, \delta \rangle$ is its i-th active edge, then:*

- $\forall j \in (1, \ldots, n)$, $\overline{q_j} \mapsto \overline{q_j}^{\widehat{i}}$,
- $\forall x \in Q$, $\forall x^{\widehat{i}} \in Q^{\widehat{i}}$, $x \mapsto x^{\widehat{i}}$ *iff* $\exists f \in Feat, \exists q \in Q, \exists q^{\widehat{i}} \in Q^{\widehat{i}}.q \mapsto q^{\widehat{i}}$, *such that* $\delta(f, q) = x$ *and* $\delta(f, q^{\widehat{i}}) = x^{\widehat{i}}$.

The mapping from nodes of $R^{\widehat{i}}$ to nodes of R associates a node in $R^{\widehat{i}}$ with a set of nodes in R:

Definition 4.4. *If $R = \langle Q, \overline{Q}, \theta, \delta \rangle$, $R^{\widehat{i}} = \langle Q^{\widehat{i}}, \overline{Q}^{\widehat{i}}, \theta, \delta \rangle$, and $x^{\widehat{i}} \in Q^{\widehat{i}}$, then the set of nodes mappable to $x^{\widehat{i}}$ is $[x^{\widehat{i}}]^{-1}$, the largest set $\{x \in Q | x \mapsto x^{\widehat{i}}\}$. The set of nodes mappable to a set of nodes $Q^{\widehat{i}'} \subseteq Q^{\widehat{i}}$ is $[Q^{\widehat{i}'}]^{-1}$, the largest set $\{x \in Q | \exists x^{\widehat{i}'} \in Q^{\widehat{i}'}$ such that $x \mapsto x^{\widehat{i}'}\}$.*

An important relation between $\mapsto$ and type subsumption exists:

Proposition 4.1. *If $R^{\widehat{i}} = \langle Q^{\widehat{i}}, \overline{Q}^{\widehat{i}}, \theta, \delta \rangle$, $R^{\widehat{j}} = \langle Q^{\widehat{j}}, \overline{Q}^{\widehat{j}}, \theta, \delta \rangle$, where $0 \leq i < j \leq |\overline{Q}|$, then $\forall x^{\widehat{i}} \in [x^{\widehat{j}}]^{-1}, \theta(x^{\widehat{i}}) \sqsubseteq \theta(x^{\widehat{j}})$.*

Proof. It will be demonstrated first that $\forall i \in (1,\ldots,n)$, where $n = |\overline{Q}|$, if $\widehat{R^{i-1}} = \langle \widehat{Q^{i-1}}, \widehat{\overline{Q}^{i-1}}, \theta, \delta\rangle$ and $\widehat{R^{i}} = \langle \widehat{Q^{i}}, \widehat{\overline{Q}^{i}}, \theta, \delta\rangle$, then $\forall \widehat{x^{i-1}} \in [\widehat{x^{i}}]^{-1}, \theta(\widehat{x^{i-1}}) \sqsubseteq \theta(\widehat{x^{i}})$.

$\widehat{R^{i}} = \widehat{R^{i-1}} \sqcup \langle E, i, n\rangle$ (where E is a TFS representing an edge in the chart.) Since $\widehat{x^{i-1}} \mapsto \widehat{x^{i}}$, then $\exists \pi \in Path$, $\exists j \in (1,\ldots,n)$, such that $\delta(\pi, \widehat{\overline{q_j}^{i-1}}) = \widehat{x^{i-1}}$ and $\delta(\pi, \widehat{\overline{q_j}^{i}}) = \widehat{x^{i}}$ (where $\widehat{\overline{q_j}^{i-1}}$ and $\widehat{\overline{q_j}^{i}}$ are roots of $\widehat{R^{i-1}}$ and $\widehat{R^{i}}$ respectively.) Therefore, according to Definition 3.4, $\widehat{\overline{q_j}^{i-1}} \bowtie \widehat{\overline{q_j}^{i}}$, and thus, $\widehat{x^{i-1}} \bowtie \widehat{x^{i}}$. But according to the same definition, $\forall q \in [\widehat{x^{i}}]_{\bowtie}, \theta(\widehat{x^{i}}) \sqsupseteq \theta(q)$. Hence, $\theta(\widehat{x^{i-1}}) \sqsubseteq \theta(\widehat{x^{i}})$.

Since both $\sqsubseteq$ and $\mapsto$ are transitive, the proposition holds for any $\widehat{R^{i}}$ and $\widehat{R^{j}}$, where $0 \leq i < j \leq n$. □

4.2 The Typed Feature Structure Indexing Problem

4.2.1 Difficulties for TFSG Indexing

In TFSGs, unification itself is very costly. Indexing could be the key to efficient parsing by reducing the number of unifications while retrieving categories from the chart.

The major problem with indexing TFSGs lies in the large amount of information stored in a TFS. This makes indexing difficult for TFSG parsers, since the extraction of an index key from each category is not a trivial process. Another obstacle is that at least some of the index has to be built and maintained during parsing (unlike in databases, where the index is typically built off-line), since not all information in a TFS is available before parsing.

All these aspects of TFSG indexing, together with similar problems and their solutions in related areas, are presented in Section 4.4 and in Chapter 5.

4.2.2 Indexing Timeline

Indexing can be applied at several moments during parsing. While Sections 4.4 and 4.5 presents and comments on several approaches to indexing, this section outlines a general strategy for indexed parsing, with respect to what actions should be taken at each stage during parsing. This outline is also valid for other formalisms, such as CFGs, with appropriate simplification.

Three main stages can be identified during parsing. The first stage describes indexing actions that can be taken off-line (along with several other actions performed during compile time, as described in Section 3.2.1). The second and third stages refers to action performed at run time.

1) In the off-line phase, an analysis of grammar rules can be performed. The actual content of mothers and daughters may not be accessible, due to variables that will be instantiated during parsing. The ideal case for indexing would be having no variables – resulting in a CFG-like grammar, in which all indexing can be done at compile time. However, various sources of information, such as the type signature, appropriateness specifications, or general descriptions of mothers and daughters, can be analyzed and an appropriate indexing scheme can be specified. This phase of indexing may include determining:

(a) which daughters in which rules are sure not to unify with a specific mother, and

(b) what information can be extracted from categories during parsing that can constitute indexing keys.

It is desirable to perform as much analysis as possible at this off-line stage, since the cost of any action taken during run time prolongs the parsing time.

2) During parsing, after a rule has been completed, all variables in a mother are instantiated. This offers the possibility of further investigating the mother's content and extracting supplemental information from the mother that contributes to the indexing keys. However, the choice of such investigative actions must be carefully studied, since it might burden the parsing process.

3) While completing a rule, for each daughter a matching edge is searched in the chart. At this moment, the daughter's Active External Variables are instantiated, therefore the entire content of the daughter can be accessed, and the information identified in stage (1b) can be extracted. Also in this stage, the unification between daughters and edges takes place.

4.3 Indexed Chart Parsing

In order to complete a rule, all the rules' daughters should be found in the chart as edges. Looking for a matching edge for a daughter is accomplished by attempting unifications (matches) with edges stored in the chart, resulting in many failed unifications.

This section introduces a general indexing method that will be used as the backbone of all indexing methods introduced in this thesis. Without entering into details pertaining to a specific grammar formalisms, the mechanism for indexing a chart parser is presented. The general method outlined here can be applied to any chart-based parser.

4.3.1 General Indexing Strategy

The purpose of indexing is to reduce the time spent on failed attempts when searching for an edge in the chart. Each edge (edge's category or description) in the chart has an associated index key which uniquely identifies sets of categories that can potentially match that edge's category. When completing a rule, the chart parsing algorithm looks up edges matching a specific daughter in the chart. Instead of visiting all edges in the chart, the daughter's index key selects a restricted number of edges for traversal, thus reducing the number of unification attempts.

The passive edges added to the chart represent specializations of rules' mothers. Each time a rule is completed, its mother $\mathcal{M}$ is added to the chart according to $\mathcal{M}$'s *indexing scheme*, i.e., the set of index keys of daughters that are possible candidates for a successful unification with $\mathcal{M}$. The indexing scheme needs to be re-built only when the grammar rules or the signature change.

From an implementation point of view, the index is represented as a hash, where the hash function applied to a daughter yields the daughter's index key. Each entry in the chart has a hash associated with it. When passive edges are added to the chart, they are inserted into one or more hash entries. For an edge representing $\mathcal{M}$, the list of hash entries where it will be added is given by $\mathcal{M}$'s indexing scheme.

4.3.2 Using the Index

Each daughter is associated with a unique index key. During parsing, a specific daughter is searched for in the chart by visiting only the list of edges that have the appropriate key, thus reducing the time needed for traversing the chart. The index keys can be computed off-line (when daughters are indexed by their position), or during parsing.

4.3.3 An Example

Figure 4.1 presents an intuitive example for a very simple context-free grammar:

$S \rightarrow VP\ NP$

$VP \rightarrow V\ NP$

$NP \rightarrow N$

$NP \rightarrow ART\ N.$

For the input string "Ducks eat flies", when looking for a matching NP in the chart to complete a rule such as $VP \rightarrow V\ NP$, the non-indexed parser first attempts to match the daughter category NP with edges V and N stored in the chart. Only after these attempts fail is the right match found. The indexed parser never attempts the matches $NP = N$ and $NP = V$, since its searches will always be limited to a list containing compatible categories.

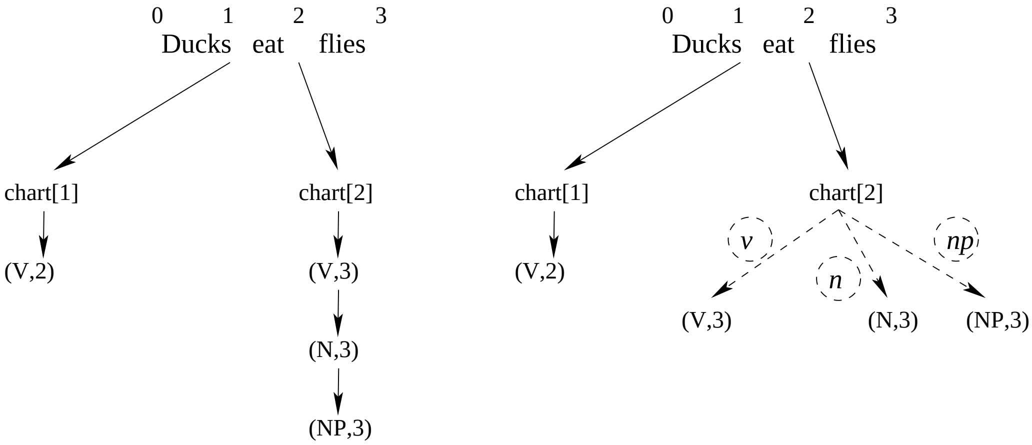


Figure 4.1: A simple example of indexed chart parsing. In this example, an entry (P, j) found in the chart location $chart[i]$ represents an edge labeled with the part of speech P and spanning from position i to position j in the input string.

This example illustrates a "perfect index" (as the one described in Section 7.4.2). In a "perfect" index, all attempted unification between a daughter and all edges in the set selected by the daughter's key would be successful. Unfortunately, TFSs contain a large amount of information, not all of which is accessible before parsing, therefore a "perfect" index is unlikely for TFSGs. However, as it will be shown in the preliminary evaluation presented in Chapter 7, even if not all possible failed unifications are avoided, indexing can still improve parsing times.

4.4 Previous Approaches to Indexing and Filtering TFSGs

As stated in Chapter 1, the goal of the methods developed in this thesis is to improve the parsing times for TFSGs, by means of indexing. At the moment of writing this thesis, the available literature on this topic is rather scarce. Most of the currently used techniques for improving TFSG parsing times use no indexing, or employ filtering as a substitute for true indexing. Other directions of research in this domain address other weaknesses of TFSG parsing, such as the inefficient encoding of typed feature structures, the parsing algorithm, or the lack of a suitable programming environment. This section presents a short overview of efforts directed at improving TFSG parsing times by means of filtering, which is the closest related technique to indexing.

4.4.1 HPSG Parsing with CFG Filtering

CFG filtering for HPSG parsing was introduced by Torisawa [1995; 2000], and it is based on the observation that many failed unifications (between edges in the chart and daughters in the grammar rules in course of completion) occur in partial parse trees that will later be discarded. By eliminating in advance partial parse trees that do not contribute to the final parse tree, a significant amount of failed unification will be avoided. This is accomplished by analyzing parse trees for an approximate CFG extracted from the HPSG.

Several methods exist for generating CFGs from UBGs. The method proposed in [Torisawa *et al.*, 2000] is an improvement over previously existing methods. The extracted CFG takes into account dependencies between rules (by analyzing feature propagations across rules), which play an important role in restricting the possible parse trees in an HPSG. The authors report improvements of up to 80% for this method. However, this result was achieved by removing features from the original grammar, making these results less realistic. More than that, the cost of filtering the CFGs is reported to be a major drawback of this method.

4.4.2 Quick Check

Perhaps the most well known method used in speeding up the parsing times of TFSGs is the quick check filter. Initially mentioned in [Kiefer *et al.*, 1999], a detailed presentation of quick check is found in [Malouf *et al.*, 2000].

Quick check is an empirical method that tries to reduce the burden placed on parsing by the unification operation. Its goal is to quickly identify, before the unification of two feature structures is attempted, whether the unification will fail.

The unification between two typed feature structures is successful if all the equivalent paths in the two TFSs are compatible. The compatibility is determined by the type unification operation between feature values at the end of these paths. Quick check relies on the empirical observation (made by parsing a corpus of sentences) that only a small amount of paths are likely to cause the majority of unification failures.

In order to identify the failure-causing paths, a training phase is required. The parser records, for each failed unification between two TFSs, the path responsible for the failure. A corpus of sentences is parsed, and a set containing the paths that are most probable to cause failures is determined.

After training, when parsing sentences in the test corpus, the parser associates each category in the grammar with a quick check vector. The vector contains the types of the feature values extracted from the ends of the paths selected during training. For a given grammar, a position i in a quick check vector will always be filled with the type extracted from the same path π_i for any feature structure. Before a unification is attempted, the quick check vectors of the two typed feature structures are matched. Only if their values are compatible is the full unification performed.

For example, assume that after training paths $F : G$ and $H : G : I$ are selected as the most probable to cause unification failure. The quick check vector is of the form $\langle v_1, v_2 \rangle$, where the type extracted from path $F : G$ will fill the value v_1 and the type extracted from path $H : G : I$ will fill the value v_2 for any feature structure. For the two feature structures presented in Figure 4.2, their corresponding quick check vectors will be $\langle t_5, t_5 \rangle$ for F_1 and $\langle \perp, t_6 \rangle$ for F_2. Before unifying F_1 and F_2, their quick check vectors are compared. The unification is attempted only if $t_5 \sqcup t_6 \downarrow$.

Quick check proves to be efficient when evaluated on a large-scale TFSG (an average improvement of 34% is reported in [Malouf *et al.*, 2000]). However, its success depends on the degree of similarity between the training corpus and the test corpus. Another requirement is that only a small number of paths should be accountable for the unification failures. It is not desirable that the the distribution of probabilities of paths to cause a failure should be flat.

$$F_1 = \begin{bmatrix} t_1 & \\ F: & \begin{bmatrix} t_2 & \\ G: & t_5 \end{bmatrix} \\ H: & \begin{bmatrix} t_3 & \\ F: & t_2 \\ G: & \begin{bmatrix} t_4 & \\ I: & t_5 \end{bmatrix} \end{bmatrix} \end{bmatrix} \qquad F_2 = \begin{bmatrix} t_1 & \\ F: & \begin{bmatrix} t_4 & \\ G: & \perp \end{bmatrix} \\ H: & \begin{bmatrix} t_3 & \\ G: & \begin{bmatrix} t_4 & \\ I: & t_6 \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

F_1 quick check vector : $\langle t_5, t_5 \rangle$ $\qquad$ F_2 quick check vector : $\langle \perp, t_6 \rangle$

Figure 4.2: Quick check vectors

4.4.3 Rule Filtering

One of the methods that inspired the work on indexing presented in Chapters 4 and 6 is rule filtering. Unfortunately, this method has received less attention as an indexing technique. The only work mentioning rule filtering is Kiefer *et al.* [1999].

Rule filtering has the same goal as the other methods presented in this section: reducing the number of failed unifications. It is based on the observation that many of these failed attempts happen during parsing while matching daughters and edges that can be ruled out before parsing — a daughter and a mother that do not unify before parsing will also not unify during parsing.

In order to rule out unifications that are sure to fail, all MGSats of daughters in the grammar are unified with all MGSats of mothers at compile time. The mother-daughter pairs that failed the unification are recorded (as a rule filtering function). During parsing, before a daughter is matched with an edge created by a mother, the rule filtering function is consulted for this mother-daughter pair, and the unification is not attempted if the pair is ruled out by the rule filter. The filter is built as a table, where each cell specify whether a particular mother and daughter can be unified. For a corpus of English, German, and Japanese sentences from Verbmobil[Kay *et al.*, 1994], the authors report that up to 60% of failed unifications can be avoided (resulting in a 45% improvement in parsing times). However, as it will be shown in Chapter 7, the percent of failed unifications that can be avoided by ruling out non-unifiable mother-daughter pairs before parsing can be much smaller for different grammars.

4.5 A Discussion About Optimization Approaches

Although there is not a significant amount of work directed at improving parsing times by use of indexing, it will be shown in this section that indexing has many advantages over filtering, which is the closest related technique. Advantages and disadvantages of using several indexing or related techniques (such as filtering) will be described, as well as a discussion about using statistically or non-statistically improved parsers.

4.5.1 Indexing vs. Filtering

Most of the research conducted on improving parsing times for TFSGs is motivated by the observation that unifying two typed feature structures is the most time-consuming operation in the parsing process. Several approaches were taken, such as improving the encoding of TFSs (in order to reduce their size or to speed up the unification process), implementing abstract machines for rule daughter matching, or filtering out unifications that are known to fail, based on fast checks before the unification is attempted. Among these, the last approach is closest to indexing.

In database systems, the *de facto* standard for retrieving entities of any type is through indexing or hashing [Elmasri and Navathe, 2000; Bertino *et al.*, 1997]. Filtering is accomplished by traversing the entire database and deciding for each entity whether it should be retrieved or not (according to a specific criterion). By indexing, only a restricted number of entities is visited. The different retrieval methods in indexing and filtering are a consequence of the fundamental difference between indexing and filtering: while indexing considers the database as a structured set, filtering assumes the database is not structured [Belkin and Croft, 1992].

The advantage of using indexing is the support for searches based on multiple criteria (an important aspect, as it will be shown in Section 6.2). Using multiple criteria with filtering means testing all criteria for each entity in the database, while in indexing, each criterion places a restriction on the search space. Another important advantage of using indexing instead of filtering is the support for complex queries that are not limited to membership checking [Manolopoulos *et al.*, 1999].

4.5.2 Statistical vs. Non-Statistical Methods

While previous sections presented an overview of several approaches to improve parsing times for TFSGs, the focus of this section in particular is on the analysis of those that use indexing or filtering. They can be divided into two major categories: statistically-based and non-statistically-based.

Statistical methods, such as the commonly used quick-check [Malouf *et al.*, 2000], need a training phase in order to determine the best criteria used to predict the unification failure. The advantage of using such methods resides in their simplicity. Although the experimental results can often exhibit significant improvements, the major disadvantage of this method is the need for a training phase. If the grammar is modified often (even through very small changes, which can occur frequently in a development process), the time spent on training is not compensated for by the improvements in parsing times. More than that, if the training set is not properly chosen, the statistical filter may even fail to detect any unification failure. Another disadvantage of methods such as quick-check, which need to determine the failure-causing paths in the training phase, is that finding optimal paths is exponential [Penn and Munteanu, 2003].

Non-statistical indexing or filtering methods used to improve parsing times have received less attention in recent years. One of the very few methods is rule filtering [Kiefer *et al.*, 1999], a method similar to the general indexing presented in Chapter 4. However, its authors do not exploit the benefits of an indexing structure, nor do they present the results in a large experimental context. As will be shown in Chapter 6, using an indexing structure allows for implementation of various extensions, such as "personalized" index keys for each mother-daughter pair (leading to a larger percentage of avoided failed unifications.)

It should be mentioned here that the non-statistical indexing methods presented later in Chapter 6 do not preclude statistical improvements[1]. Although quick-check and rule filtering demonstrate improvements on parsing times, both methods were evaluated using parsers that are not fully optimized from a non-statistical point of view. A complete evaluation of such statistical methods would be more relevant if performed after all posible non-statistical optimizations are implemented.

[1]However, the main advantage of using a non-statistical method is the absence of the training phase, resulting in faster parser set-up times.

Chapter 5

Indexing for Non-Parsing Applications

While Section 4.4 presented an overview of indexing/filtering methods used in parsing applications, this chapter presents similar techniques used in other TFS applications. Also, the current research in the area of general term indexing is outlined. Although not directly connected to the domain of TFSG parsing, the methods outlined in this chapter can provide a source of knowledge in approaching the TFSG indexing problem, by identifying differences and similarities between these domains.

5.1 Indexing For Other TFS Applications

Not only parsing with typed feature structure grammars suffers from the cost of unification. Most TFS-based applications encounter the same obstacle to efficiency. In this section, two indexing methods that alleviate the burden of unification are presented.

5.1.1 An Indexing Scheme for TFS Retrieval

Ninomiya *et al.* [2002] propose an indexing method for typed feature structure retrieval engines. Given a database of TFSs, and a TFS query, the indexed retrieval engine selects a reduced number of TFSs for unification with the TFS query.

The index is built as a table with a row for each possible path in a TFS. The columns represent types (feature values). Each position (cell) indicated by a row π and column t in the table contains a list of TFSs from the database that have a type compatible with t at the end of the path π. When a TFS query is given, the table entry for which a path is defined in the TFS query is selected, and the corresponding feature value is extracted from the TFS query. If there are more than one table entries that can be selected, the one containing the shortest list of TFSs is chosen. Only TFSs indicated by the matching list are selected for unification with the TFS query. Although the reported improvements in TFS retrieval times reach 37%, the costs of building such an index table could be prohibitive for TFSG parsing.

5.1.2 Automaton-based Indexing for Lexical Generation

An interesting indexing method is proposed in [Penn and Popescu, 1997], employed for lexical generation (surface realization) in ALE [Carpenter and Penn, 2001]. An automaton-based index is used to extract words from the lexicon with descriptions matching a given typed feature structure.

A decision tree is built off-line, with nodes representing features and arcs representing types. The leaves of the tree point to lexical entries. The feature paths that are indexed are determined manually as paths that reach values bearing a semantic content. Each time a word is searched for by a feature structure, the indexing path is followed in the decision tree, and the reached leaf will give the desired lexical entry. Unfortunately, this method cannot be applied to parsing. In generation, the index can be entirely built off-line, since all paths in the feature structures representing lexical entries are fully accessible off-line. In parsing, it is not lexical entries that must be indexed, but categories in the grammar.

5.2 General Term Indexing

UBGs are just one area of artificial intelligence where indexing can be used to improve efficiency. In general, any application dealing with large knowledge bases could benefit from indexing. Such applications include automated reasoning systems, symbolic computing systems, or term rewriting applications. In these applications, the knowledge bases can consist of first-order terms, clauses, or formulae. Parsing with TFSGs can be seen as a particular case of such applications.

Term indexing serves the purpose of rapidly retrieving candidate terms that satisfy a specific property. Formally, the term indexing problem can be formulated [Ramakrishnan *et al.*, 2001] as: Given a set of indexed terms L and a binary relation (the retrieval condition, or indexing function) over terms R, for a query term t determine the subset $M \subset L$ such that $M = \{s|R(s,t)\downarrow\}$.

The following subsections outline various term indexing methods. The extensive presentations of term indexing found in [Ramakrishnan *et al.*, 2001] and [Graf, 1996] are used here as a guide.

5.2.1 Attribute-Based Indexing

Attribute-based indexing uses simple values to map features of terms into attributes. The retrieval of terms is performed based on the relation existing between the corresponding attributes. Even if this method is very simple to implement, its low accuracy is a disadvantage compared to more complex techniques.

5.2.2 Set-Based Indexing

Set-based indexing, as the name suggests, divides the set of indexed terms into subsets, such that all terms in a subset have a common property (an individual term can be placed in several subsets).

The most simple set-based indexing – *top symbol hashing* – can be found in Prolog as first argument indexing. Clauses are indexed (hashed) using their first argument, therefore using the first argument of a calling procedure, a set of possible matching candidate clauses is retrieved from the assertional database.

A more complex indexing method is *path indexing*. Paths leading to symbols inside a term are grouped into sets (path lists) based on the common properties that are shared between the paths. Paths in the query term are organized into a query tree that has leaves pointing to the path lists. The retrieval of terms is accomplished by traversing the paths in the query tree.

5.2.3 Tree-Based Indexing

Tree-based indexing organizes the terms into a single tree. Each path into the tree represents common properties of the indexed terms, similar to decision trees or classification trees.

The basic tree-based indexing method is *discrimination tree* indexing. The tree reflects exactly the structure of terms. A more complex tree-based method is *abstraction tree* indexing. The nodes are labeled with lists of terms, in a manner that reflects the substitution of variables from a term to another: the domain of variable substitutions in a node is the codomain of the substitutions in a subnode (substitutions are mappings from variables to terms).

A relatively recent tree-based method was proposed in [Graf, 1995]: *substitution tree indexing.* This is an improved version of discrimination tree and abstraction tree indexing. Each path in the tree represents a chain of variable bindings. The retrieval of terms is based on a backtracking mechanism similar to the one in Prolog. Substitution tree indexing exhibits retrieval and deletion times faster than other tree-based indexing methods. However, it has the disadvantage of slow insertion times.

Since typed feature structures can be viewed as similar to first order terms with variables, the unification process requires a sequence of substitutions. Substitution tree indexing could be applied to TFSGs; unfortunately, published experimental results [Graf, 1995] indicating slow insertion times suggest that a method performing more efficient operations during run time is to be preferred. Future work will investigate possible adaptations of this technique to TFSG parsing.

5.3 Indexing in Database Systems

Although database systems are not in the scope of this thesis, many of the techniques developed here are connected to the database area. Since the subject of indexing in databases is very vast, just a few essential bibliographical pointers are mentioned in this section.

Databases can store large amounts of data. Usually, each stored entity is a complex structure, called a record (similar to a feature structure). Records are indexed based on the values of certain fields (features). The retrieval is usually not limited to a query where specific values are requested for a field, but must support other types of queries (such as interval queries – where the values should belong to a given interval). An interesting research topic in the area of indexing are the self-adaptive indexing methods, where the indexing can be (semi-)automatically configured. One of the first published work on this topic is [Hammer and Chan, 1976].

Most of the available database textbooks (such as [Elmasri and Navathe, 2000]) have chapters dedicated to indexing. Recent research papers on indexing can be found in Kluwer Academic Publishers' series "Advances in Database Systems": [Bertino *et al.*, 1997], [Manolopoulos *et al.*, 1999], or [Mueck and Polaschek, 1997].

A major difference between indexing in databases and indexing in a TFSG parser should be noted. Typically, a database consists of a large collection of objects, and the indexing scheme is designed to improve retrieval times. It is expected that databases are persistent, with fewer deletions and insertions than retrievals. From this point of view, parsing can be seen as managing a volatile database, that is always empty at start-up. The ratio between insertions and retrievals in a database application is very small (even equal to 0 when used only to retrieve data). For indexed parsing, this ratio is much higher and depends on the structure of grammar rules. For this reason (similar to those discussed in Section 5.2.3), indexing methods such as B-trees (commonly used in databases), where the retrieval can be performed in $O(1)$ operations, but the insertion needs $O(log_m(n))$ operations [Ramesh *et al.*, 2001] (where n is the number of nodes in the B-tree and m the number of index keys assigned to a node), are not recommended for indexed parsing.

Chapter 6

TFSG Indexing through Static Analysis

This chapter presents two approaches to indexing TFSG parsers. Both methods introduced here follow the general strategy outlined in Section 4.3.1 and the indexing timeline introduced in Section 4.2.2. Both methods are non-statistical indexing methods (the index is determined without parsing a training corpus). The first indexing method (*positional indexing*) uses daughters' positions in grammar rules (the rule number and daughter position in the rule) as index keys. The second method (*path indexing*) is built on top of positional indexing. Its index keys are enhanced with information extracted from mothers and daughters, information that is obtained through static analysis of grammar rules, as will be shown in Section 6.2.1.

6.1 Positional Indexing

In the indexing method presented in this section the index is computed off-line, without parsing a training corpus. The index key for each daughter is represented by its position (rule number and daughter position in the rule, hence the name "positional" indexing), therefore no time is spent during parsing for computing the index keys.

A related method (rule filtering) was proposed in [Kiefer *et al.*, 1999] (described in Section 4.4.3). However, the authors use a filter to avoid unsuccessful unifications. Using an index offers the advantages of a flexible, yet organized approach: as will be shown over the next sections, the indexing scheme developed here can be enhanced with information extracted from further static analysis of the grammar rules, while maintaining the same general indexing strategy.

6.1.1 Building the Index

The indexing method introduced in this section follows the general strategy outlined in Section 4.3. Each mother is inserted as an edge in the indexed chart only in positions specified by that mother's indexing scheme (which is the list of matching daughters' index keys). When completing a rule, a matching edge is searched for each daughter only in the chart entry labeled with an index key compatible with that daughter's index key.

The structure of the index can be determined at compile-time. The first step is to create a list containing the descriptions of all rules' mothers in the grammar. Then, for each mother description, a list

$L(Mother) = \{(R_i, D_j)|$ daughters that can match $Mother\}$ is created, where each element of the list L represents the rule number R_i and daughter position D_j inside rule R_i $(1 \leq j \leq arity(R_i))$ of a category that can match with $Mother$.

For UBGs it is not possible to determine the exact list of matches between daughters and mothers, since there are sometimes infinitely many possible variants of daughters in a given signature. However, it is possible to rule out MGSats of daughters that are incompatible (with respect to unification) with a certain $Mother$ before parsing. For the 17 mothers in the grammar used for the experiments presented in Chapter 7, the number of matching daughters statically determined before parsing ranges from 30 (the total number of daughters in the grammar) to 2. This compromise pays off with its simplicity, which is reflected in the time spent managing the index.

During run-time, each time an edge (representing a rule's mother) is added to the chart, its category is inserted into the corresponding hash entries associated with the positions (R_i, D_j) from the list $L(Mother)$ (the number of entries where $Mother$ is inserted is equal to the cardinality of $L(Mother)$). The entry associated to the key (R_i, D_j) will contain only categories (based on $MGSat(Mother)$) that can possibly unify with the daughter at position (R_i, D_j) in the grammar.

It should be mentioned at this point that only daughters D_i with $i \geq 2$ are searched for in the chart (and consequently, indexed). As described in Section 4.1.3, whenever an edge is added to the chart, all rules are visited in a failure-driven loop. The rules of which the first (leftmost) daughter unifies with the edge are then traversed left to right, starting with the second daughter.

6.1.2 Using the Index

For TFSGs, using a positional index key for each daughter presents the advantage of not needing an indexing (hash) function during parsing for the TFSs themselves (a discussion of when positional indexing is not a good solution is given in Section 7.4.2). When a rule is extended during parsing, a matching edge for each daughter is looked up in the chart. The position of the daughter (R_i, D_j) acts as the index key, and matching edges are searched for only in the list indicated by the key (R_i, D_j).

6.2 Path Indexing

Path indexing is an extension of the positional indexing presented in Section 6.1. Its functionality is related to quick check: extract a vector of types (of feature values) from a mother (that will become and edge) and from a daughter, and test the unification of the two vectors before attempting to unify the edge and the daughter.

Path indexing differs from quick-check in two major aspects. First, as mentioned in Section 4.5.2, quick check needs statistical training to decide from which nodes to extract the quick check vectors. Path indexing identifies these nodes by a static analysis of grammar rules, performed off-line and with no training required. Second, path indexing is built on top of positional indexing, therefore the vector of nodes used as a pre-test for unification can be different for each pair of mother-daughter that can unify. This does not appear to be the case for quick-check, as it was conceived of in [Malouf *et al.*, 2000].

This section will first introduce the theoretical foundations of the static analysis used to select nodes for inclusion in the path indexing. Next, details about path indexing will be given.

6.2.1 Static Analysis of Grammar Rules

The positional indexing introduced in Section 6.1 has the benefit of allowing for a simple, yet efficient, implementation. Section 7.4.2 discuss the situations when even much simpler indexing schemes are possible. However, its major advantage is its flexibility. Since each daughter is allocated a separate entry in the chart, further information collected about the peculiarities of each mother-daughter pair (with respect to the unification of the respective mother and daughter) can be integrated into the indexing scheme. This is one of the fundamental differences between indexing and filtering, and another reason for preferring indexing over filtering.

As indicated in the indexing timeline introduced in Section 4.2.2, three points during parsing can be identified for indexing. In the first, a static analysis of grammar rules can determine what information can be extracted from categories that can be used as indexing keys. At this point, the *Static Cut* will be determined for each mother and daughter that are unifiable, and index keys will be identified through the Static Cut. During parsing, the second point occurs at the completion of a rule. At this moment, its mother is introduced as an edge into the chart. The *Dynamic Cut* is used here to further refine the index keys. The third point (occurring when a matching edge for a daughter is searched for in the chart) involves the extraction of the daughter's index key.

The Static Cut

The Static Cut defines nodes in a mother[1] that carry no relevant information with respect to the unification with a daughter. Intuitively, these nodes can be "left out" while computing the unification.

Definition 6.1. *For a mother M and a daughter D that are unifiable before parsing, the static cut is defined as* $StaticCut(M, D) = RigidCut(M, D) \cup VariableCut(M, D)$.

The $RigidCut$ is defined first, representing nodes that can be "left out" because neither they, nor one of their ancestors, can have their type values changed by means of external sharing.

Definition 6.2. $RigidCut(M, D) = \{x \in Q_M | \{[y]_{\bowtie} \in Q_{M \sqcup D} | \exists \pi . \delta^*_{M \sqcup D}(\pi, [y]_{\bowtie}) = [x]_{\bowtie}\} \cap Ext(M \sqcup D) = \emptyset\}$, *where* $\bowtie$ *is the equivalence relation from the definition of typed feature structure unification with respect to* $M \sqcup D$.

The $VariableCut$ is now defined as nodes that are either externally shared, or have an ancestor that is externally shared, but still can be "left out". The name *VariableCut* is inspired by the fact that variables in descriptions are the source for external sharing between most general satisfiers, as described in Section 3.2.2.

[1] Throughout this section, mother M and daughter D are TFSs and denote the MGSat of a mother description and of a daughter description.

Definition 6.3. *$VariableCut(M, D)$ is the largest subset of $\{x \in Q_M\}$ such that:*

1) $x \notin RigidCut(M, D)$

2) $\forall s \sqsupseteq \theta(x), \forall t \sqsupseteq \theta([x]_{\bowtie} \cap Q_D).s \sqcup t \downarrow$

Basically, definition 6.3 is saying that a node can be left out even if it is externally shared (or has an externally shared ancestor) if all possible types this node can have unify with all possible types its corresponding nodes in $M \sqcup D$ can have. The first condition can also be written as follows: $\exists y \in Ext(M \sqcup D).\exists \pi.\delta^*_{M \sqcup D}(\pi, [y]_{\bowtie}) = [x]_{\bowtie}$.

Definition 6.4. *$M^{\rtimes} = \langle Q', \overline{Q'}, \theta, \delta\rangle$ is the Statically Cut typed feature structure (symbolized as $M^{\rtimes_D}$) of a mother $M = \langle Q, \overline{q}, \theta, \delta\rangle$ with respect to a daughter D iff $Q' = Q \setminus StaticCut(M, D)$.*

It should be mentioned that $M^{\rtimes_D}$ is created after the mother M is created (i.e., after the rule is completed). Only the nodes in M that are accessible before parsing can be included in the $RigidCut$ or in the $VariableCut$.

Intuitively, the nodes in the Statically Cut typed feature structure of M after the rule is completed (symbolized as $\widehat{M^{\rtimes_D}}$) are mappings of the nodes in $M^{\rtimes_D}$ before rule is completed.

Definition 6.5. *$\widehat{M^{\rtimes_D}} = \langle Q', \overline{Q'}, \theta, \delta\rangle$ is the Statically Cut typed feature structure of a mother $M = \langle Q, \overline{q}, \theta, \delta\rangle$ with respect to a daughter D after M's rule is completed iff*

- *M's rule represented as an MRS is $R = \langle \ldots, M\rangle$, and $M^{\widehat{n}}$ is the mother of the same rule after the rule is completed: $R^{\widehat{n}} = \langle \ldots, M^{\widehat{n}}\rangle$ (where n is the length of R),*

- Q' *is the largest set of nodes* $\{\widehat{x} \in Q_{M^{\widehat{n}}} | \exists x \in ([\widehat{x}]^{-1} \cap Q), x \notin StaticCut(M, D)\}$.

In the above definition, the set $[\widehat{x}]^{-1}$ must be intersected with Q in order to exclude from Q' daughters' nodes that, by means of structure sharing, are mapped into $\widehat{x}$ during parsing.

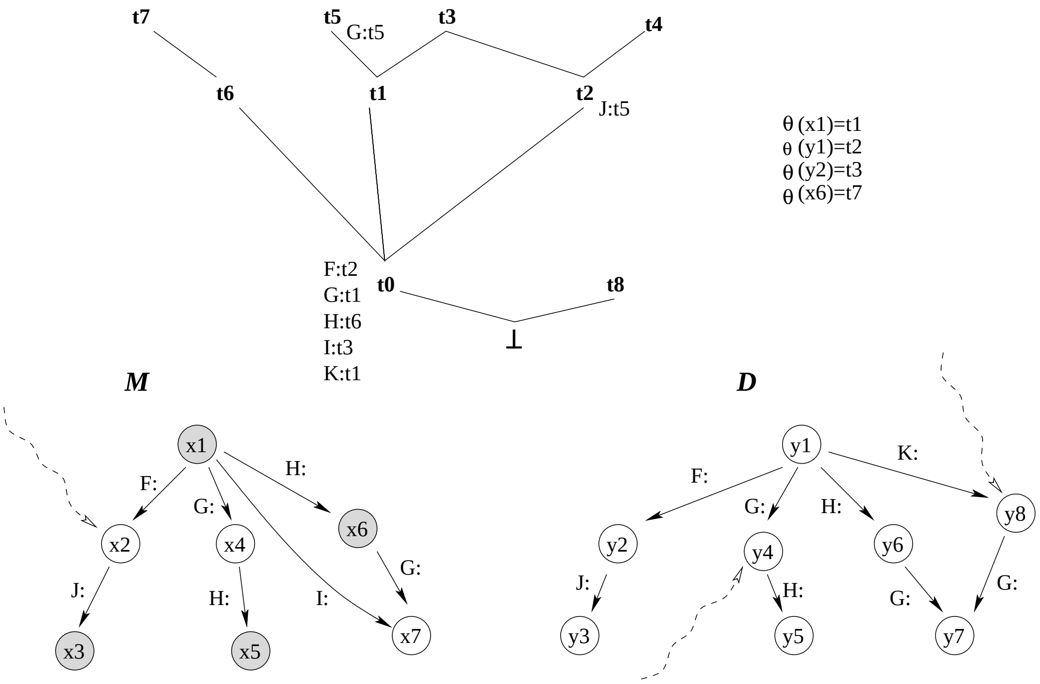


Figure 6.1: Static Cut – An Example. The dotted lines pointing to nodes x_2, y_3, and y_7 represents external structure sharing (caused by active external variables.) After the static analysis is performed, $StaticCut(M, D) = \{x_1, x_3, x_5, x_6\}$.

Figure 6.1 presents an example of a mother M and a daughter D that are unifiable before parsing. Nodes x_1, x_6, y_1 and y_2 have the following types assigned initially: $\theta(x_1) = t_1, \theta(x_6) = t_7, \theta(y_1) = t_2, \theta(y_2) = t_3$. Type values for the rest of the nodes are drawn from the appropriateness specifications: $\theta(x_2) = t_2, \theta(x_3) = t_5, \theta(x_4) = t_1, \theta(x_5) = t_6, \theta(x_7) = t_3, \theta(y_3) = t_5, \theta(y_4) = t_1, \theta(y_5) = t_6, \theta(y_6) = t_6, \theta(y_7) = t_1, \theta(y_8) = t_1$. The following equivalence relations exist between nodes from M and from D: $x_1 \bowtie y_1$, $x_2 \bowtie y_2$, $x_3 \bowtie y_3$, $x_4 \bowtie y_4$, $x_5 \bowtie y_5$, $x_6 \bowtie y_6$ $x_7 \bowtie y_7$.

Given the type hierarchy, the initial type assignments, the types from appropriateness, and the equivalence relations between nodes from M and from D, the static analysis on M and D produces $StaticCut(M, D) = \{x_1, x_3, x_5, x_6\}$. For each node in M, the static analysis produced the results as follows:

x_1: Since there is no external structure sharing on x_1 or y_1, $x_1 \in RigidCut(M, D)$.

x_2: There is external structure sharing on x_2. $\theta(x_2) = t_2$ (through appropriateness specification for feature F), and $\theta(y_2) = t_3$. Since t_2 has t_4 as a subtype, Condition 2 of Definition 6.3 is not satisfied ($t_4 \sqcup t_3 \uparrow$). Therefore, $x_2 \notin VariableCut(M, D)$.

x_3: $x_3 \notin RigidCut(M, D)$ since its ancestor x_2 is in an external structure sharing. However, $x_3 \in VariableCut(M, D)$ since Condition 2 is satisfied for x_3 and y_3 ($\theta(x_3) = t_5$ and $\theta(y_3) = t_5$.)

x_4: Similar to x_2, $x_4 \notin StaticCut(M, D)$, with the difference that the external sharing is on D's side (on node y_4.)

x_5: Condition 2 is satisfied for x_5 and y_5 ($\theta(x_5) = t_6$ and $\theta(y_5) = t_6$), therefore $x_5 \in VariableCut(M, D)$.

x_6**:** No external sharing is on x_6 or y_6 (or any of their ancestors), therefore $x_6 \in \mathit{RigidCut}(M, D)$.

x_7**:** The most interesting example is x_7. Even though its ancestors have no external sharing, its corresponding y_7 ($x_7 \bowtie y_7$) does, by being shared between two paths (one of them being $K : G$, which does not appear in M). $\theta(x_7) = t_3$ (by unifying the appropriateness for I and G) and $\theta(y_7) = t_1$. Since $t_5 \sqsupseteq t_1$, and $t_5 \sqcup t_3 \uparrow$, Condition 2 fails, and therefore, $x_7 \notin \mathit{StaticCut}(M, D)$. This is indeed justified, because $\theta(y_8)$ can promote from t_1 to any subtype of t_1, including t_5. But for t_5, appropriateness promotes the type restriction of feature G to t_5, thus promoting $\theta(y_7)$ from t_1 to t_5, which no longer unifies with $\theta(x_7)$.

Proposition 6.1. *For a mother M and a daughter D, if $M \sqcup D \downarrow$ before parsing, and $\widehat{M}$ and $\widehat{D}$ exist (after completion), then after completion:*

1) $\widehat{M}^{\scissors D} \sqcup \widehat{D} \downarrow \Rightarrow \widehat{M} \sqcup \widehat{D} \downarrow$,

2) $\widehat{M}^{\scissors D} \sqcup \widehat{D} \uparrow \Rightarrow \widehat{M} \sqcup \widehat{D} \uparrow$.

In the above proposition, the notation $\widehat{D}$ symbolizes the daughter $\widehat{D_i^{i-1}}$ in a rule $\widehat{R^{i-1}} = \langle \widehat{D_1^{i-1}}, \ldots, \widehat{D_i^{i-1}}, \ldots \rangle$ (i.e., the rule $R = \langle D_1, \ldots, D_i, \ldots \rangle$ during completion, after daughters $D_1, \ldots, D_{i-1}$ are unified with edges from the chart.)

Proof. Follows from the proof of Proposition 6.6. $\square$

The Dynamic Cut

In the following definitions, the notion of *DynamicCut* is introduced. This refers to nodes in the mother M after the grammar rule is completed, and mother M is ready to be inserted in the chart as an edge $\widehat{M}$.

Definition 6.6. *For a mother M and a daughter D that are unifiable before parsing, the dynamic cut $DynamicCut(\widehat{M}, D)$ is defined as the largest subset of $\{\widehat{x} \in Q^{\widehat{M}}\}$ such that:*

1) $\exists x \in [\widehat{x}]^{-1}$, $x \notin StaticCut(M, D)$,

2) $\forall y \in Q_D$, if $\exists x \in [\widehat{x}]^{-1}$. $x \bowtie y$, then $\forall s \sqsupseteq \theta(\widehat{x}), \forall t \sqsupseteq \theta(y), s \sqcup t \downarrow$.

Note: If condition 2 in Definition 6.6 is not satisfied, the newly created edge from mother $\widehat{M}$ may no longer unify with daughter D.

Definition 6.7. $\widehat{M}^{\text{✂}D}_{\text{✂}} = \langle Q', \overline{Q'}, \theta, \delta \rangle$ *is the Dynamically Cut typed feature structure of a mother $M = \langle Q, \overline{q}, \theta, \delta \rangle$ with respect to a daughter D iff*

- *M's rule represented as an MRS is $R = \langle \ldots, M \rangle$, and $\widehat{M} = M^{\widehat{n}}$ is the mother of the same rule after the rule is completed: $R^{\widehat{n}} = \langle \ldots, M^{\widehat{n}} \rangle$ (where n is the length of R),*

- *Q' is the largest set of nodes $\{\widehat{x} \in (Q_{\widehat{M}} \setminus DynamicCut(\widehat{M}, D)) | \forall x \in ([\widehat{x}]^{-1} \cap Q), x \notin StaticCut(M, D)\}$.*

Before stating the main proposition about the Dynamic Cut, some elementary propositions are necessary for establishing correctness. The shorter notations $\widehat{M}$ and $\widehat{D}$ will be used here to symbolize $Q_{\widehat{M}}$ and $Q_{\widehat{D}}$, while M and D symbolize Q_M and Q_D.

Proposition 6.2. *$\forall \widehat{x} \in \widehat{M}$ such that $\widehat{x} \notin \widehat{M}^{\prec_D}$, if there exists $x \in M$ such that $x \mapsto \widehat{x}$ and $x \in RigidCut(M, D)$, then $[\widehat{x}]^{-1} = \{x\}$.*

Proof. By Definition 6.2, there are no externally shared nodes on any path leading to x. Therefore, all paths in $\widehat{M}$ leading to $\widehat{x}$ exists also in M and lead to x. $\square$

Corollary 6.1. *If $\widehat{x} \in \widehat{M} \cup \widehat{D}$ and $|[\widehat{x}]_{\bowtie}| > 1$, then $[[\widehat{x}]_{\bowtie}]^{-1} \cap RigidCut(M, D) = \emptyset$.*

Corollary 6.2. *$\forall \widehat{x} \in \widehat{M}$ such that $\widehat{x} \notin \widehat{M}^{\prec_D}$, if $\{x_1, x_2\} \subseteq [\widehat{x}]^{-1}$, $x_1 \neq x_2$, then $x_1, x_2 \in VariableCut(M, D)$.*

Proposition 6.3. *If $\widehat{x} \in \widehat{M}$ and $|[\widehat{x}]_{\bowtie} \cap \widehat{M}| = 1$ ($\bowtie$ with respect to $\widehat{M} \sqcup \widehat{D}$), then $\forall \widehat{y} \in [\widehat{x}]_{\bowtie} \cap \widehat{D}$, $\exists x \in [\widehat{x}]^{-1}$, $\exists y \in [\widehat{y}]^{-1}$ such that $x \bowtie y$ ($\bowtie$ with respect to $M \sqcup D$). Similarly, if $\widehat{y} \in \widehat{D}$ and $|[\widehat{y}]_{\bowtie} \cap \widehat{D}| = 1$ ($\bowtie$ with respect to $\widehat{M} \sqcup \widehat{D}$), then $\forall \widehat{x} \in [\widehat{y}]_{\bowtie} \cap \widehat{M}$, $\exists y \in [\widehat{y}]^{-1}$, $\exists x \in [\widehat{x}]^{-1}$ such that $y \bowtie x$ ($\bowtie$ with respect to $M \sqcup D$).*

Proof. $\forall \widehat{y}, \widehat{y'} \in [\widehat{x}]_{\bowtie} \cap \widehat{D} (\widehat{y} \neq \widehat{y'})$, there is no path in $\widehat{D}$ leading to both $\widehat{y}$ and $\widehat{y'}$ (otherwise $\widehat{y} = \widehat{y'}$). Thus, $\widehat{y} \bowtie \widehat{y'}$ because of the transitivity of $\bowtie$ ($\bowtie$ with respect to $\widehat{M} \sqcup \widehat{D}$). Therefore, since $|[\widehat{x}]_{\bowtie} \cap \widehat{M}| = 1$, for any $\widehat{y} \in [\widehat{x}]_{\bowtie} \cap \widehat{D}$ there is at least one path π leading to $\widehat{x}$ in $\widehat{M}$ and to $\widehat{y}$ in $\widehat{D}$. Therefore, according to the definition of $\mapsto$, $\exists x \in [\widehat{x}]^{-1}, y \in [\widehat{y}]^{-1}$ such that $x \bowtie y$ ($\bowtie$ with respect to $M \sqcup D$). $\square$

Figure 6.2 presents an example of a situation where this Proposition holds. As can be seen in the figure, $[\widehat{x}]^{-1} = \{x_1, x_2\}$, $[\widehat{y}]^{-1} = \{y_1\}$, $[\widehat{y'}]^{-1} = \{y_2, y_3\}$, $[\widehat{x}]_{\bowtie} \cap \widehat{M} = \{\widehat{x}\}$, and $[\widehat{x}]_{\bowtie} \cap \widehat{D} = \{\widehat{y}, \widehat{y'}\}$. Proposition 6.3 states that for both $\widehat{y}$

and $\widehat{y}'$ there is at least one node in D mappable to them which is equivalent to at least one node in M that maps to $\widehat{x}$. Indeed, for $\widehat{y}$ it is $y_1 \in [\widehat{y}]^{-1}$ that is equivalent to $x_1 \in [\widehat{x}]^{-1}$ ($x_1 \bowtie y_1$), and for $\widehat{y}'$ it is $y_2 \in [\widehat{y}']^{-1}$ that is equivalent to $x_2 \in [\widehat{x}]^{-1}$ ($x_2 \bowtie y_2$).

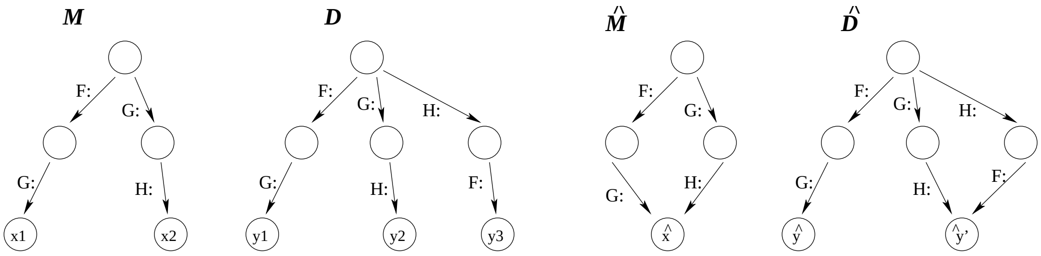


Figure 6.2: An example of the applicability of Proposition 6.3.

It should be noted that in the above proposition, $[\widehat{x}]_{\bowtie} \cap \widehat{M}$ is restricted to a single element, and thus, the proposition states the existence of $y \in [\widehat{y}]^{-1}$ equivalent to $x \in [\widehat{x}]^{-1}$ and not to $x \in [[\widehat{x}]_{\bowtie}]^{-1}$. Figure 6.3 presents an example where $[\widehat{x_1}]_{\bowtie} \cap \widehat{M} = \{\widehat{x_1}, \widehat{x_2}\}$ and $[\widehat{x_1}]_{\bowtie} \cap \widehat{D} = \{\widehat{y_1}, \widehat{y_2}, \widehat{y_4}\}$. $\widehat{x_1} \bowtie \widehat{y_4}$, $[\widehat{y_4}]^{-1} = \{y_4\}$, and $[\widehat{x_1}]^{-1} = \{x_1, x_2\}$; however neither x_1 nor x_2 are equivalent to y_4.

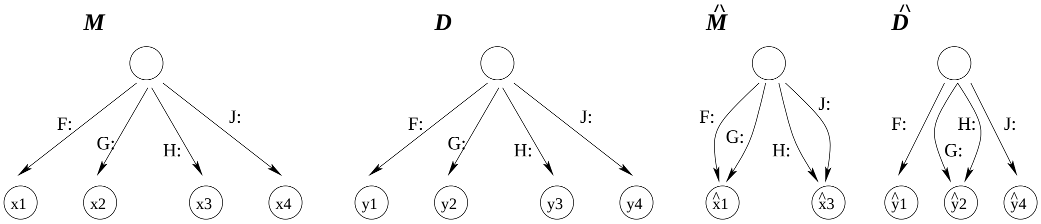


Figure 6.3: An example of Proposition 6.3 limitation to $|[\widehat{x}]_{\bowtie} \cap \widehat{M}| = 1$.

Proposition 6.4. *If $x \in M, x \in RigidCut(M, D), y \in D$ such that $x \bowtie y$ and $x \mapsto \widehat{x}, y \mapsto \widehat{y}$, then $\theta(\widehat{x}) = \theta(x)$ and $\theta(\widehat{y}) = \theta(y)$.*

Proof. By Definition 6.2, there are no externally shared nodes on any path leading to x or y. Therefore, during parsing, no types of any of x or y's ancestor nodes is promoted, and thus, $\theta(\widehat{x}) = \theta(x)$ and $\theta(\widehat{y}) = \theta(y)$. $\square$

Proposition 6.5. *If $t_0, t_1, \ldots, t_n \in Type$ such that $\forall t'_0 \sqsupseteq t_0, \forall i \in (1, \ldots, n)$, $t'_0 \sqcup t_i \downarrow$, then $\exists t \sqsupseteq t_0$ such that $\forall i \in (1, \ldots, n)$, $t \sqsupseteq t_i$.*

Proof. By induction on n $(n \geq 1)$.

Base case $(n = 1)$: $\forall t'_0 \sqsupseteq t_0, t'_0 \sqcup t_1 \downarrow \Rightarrow t_0 \sqcup t_1 \downarrow \Rightarrow \exists t \sqsupseteq t_0$ such that $t \sqsupseteq t_1$.

Induction $(n > 1)$: Assume the proposition holds for $n - 1$: $\exists t' \sqsupseteq t_0$ such that $\forall i \in (1, \ldots, n-1), t' \sqsupseteq t_i$. If $\forall t'_0 \sqsupseteq t_0, t'_0 \sqcup t_n \downarrow$, then also $t' \sqcup t_n \downarrow$. Therefore, $\exists t \sqsupseteq t'$ such that $t \sqsupseteq t_n$ and thus, $\exists t \sqsupseteq t_0$ such that $\forall i \in (1, \ldots, n), t \sqsupseteq t_i$.

$\square$

Proposition 6.6. *For a mother M and a daughter D, if $M \sqcup D \downarrow$ before parsing, and $\widehat{M}$ and $\widehat{D}$ exist (after completion), then after completion:*

1) $\widehat{M}^{\scissors D}_{\scissors} \sqcup \widehat{D} \downarrow \Rightarrow \widehat{M} \sqcup \widehat{D} \downarrow$,

2) $\widehat{M}^{\scissors D}_{\scissors} \sqcup \widehat{D} \uparrow \Rightarrow \widehat{M} \sqcup \widehat{D} \uparrow$.

Proof. The first part of the proposition, if $\widehat{M}^{\scissors D}_{\scissors} \sqcup \widehat{D} \downarrow$, then $\widehat{M} \sqcup \widehat{D} \downarrow$ will be proven by showing that $\forall \widehat{z} \in \widehat{M} \cup \widehat{D}$, $\theta^{\bowtie}([\widehat{z}]_{\bowtie}) \downarrow$. The shorter notations $\widehat{M}$ and $\widehat{D}$ will be used here to symbolize $Q_{\widehat{M}}$ and $Q_{\widehat{D}}$.

<u>**Part 1**</u> Four cases can be identified:

Case A: $|[\widehat{z}]_{\bowtie} \cap \widehat{M}| = 1$ and $|[\widehat{z}]_{\bowtie} \cap \widehat{D}| = 1$,

Case B: $|[\widehat{z}]_{\bowtie} \cap \widehat{M}| = 1$ and $|[\widehat{z}]_{\bowtie} \cap \widehat{D}| > 1$,

Case C: $|[\widehat{z}]_{\bowtie} \cap \widehat{M}| > 1$ and $|[\widehat{z}]_{\bowtie} \cap \widehat{D}| = 1$,

Case D: $|[\widehat{z}]_{\bowtie} \cap \widehat{M}| > 1$ and $|[\widehat{z}]_{\bowtie} \cap \widehat{D}| > 1$,

Case E: $|[\widehat{z}]_{\bowtie} \cap \widehat{M}| = 0$ or $|[\widehat{z}]_{\bowtie} \cap \widehat{D}| = 0$,

Both of the cases A and B can have four subcases, based on where the node $\widehat{x}$ ($\{\widehat{x}\} = [\widehat{z}]_{\bowtie} \cap \widehat{M}$) belongs to:

Case i: $\widehat{x} \in \widehat{M}$, $\widehat{x} \notin \widehat{M}^{\scissors_D}$, and $([[\widehat{x}]_{\bowtie}]^{-1} \cap M) \subseteq RigidCut(M, D)$,

Case ii: $\widehat{x} \in \widehat{M}$, $\widehat{x} \notin \widehat{M}^{\scissors_D}$, and $([[\widehat{x}]_{\bowtie}]^{-1} \cap M) \subseteq VariableCut(M, D)$,

Case iii: $\widehat{x} \in \widehat{M}$, $\widehat{x} \in \widehat{M}^{\scissors_D}$, $\widehat{x} \notin \widehat{M}^{\scissors_D}_{\scissors}$ and $\widehat{x} \in DynamicCut(\widehat{M}, D)$,

Case iv: $\widehat{x} \in \widehat{M}$, $\widehat{x} \in \widehat{M}^{\scissors_D}$, $\widehat{x} \in \widehat{M}^{\scissors_D}_{\scissors}$.

For cases C and D, the above subcase i is not possible (according to Corollary 6.1), and subcases ii – iv will be treated as a single, unified, subcase ($\widehat{z} \in \widehat{M}$).

<u>**Case A.**</u> It will be shown that $\theta^{\bowtie}([\widehat{z}]_{\bowtie}) \downarrow$ by showing that $\theta(\widehat{x}) \sqcup \theta(\widehat{y}) \downarrow$, where $\{\widehat{x}\} = [\widehat{z}]_{\bowtie} \cap \widehat{M}$ and $\{\widehat{y}\} = [\widehat{z}]_{\bowtie} \cap \widehat{D}$.

<u>Case A.i.</u> $[\widehat{x}]^{-1} \subseteq RigidCut(M, D)$. Proposition 6.3 states that $\exists x \in [\widehat{x}]^{-1}, \exists y \in [\widehat{y}]^{-1}$ such that $x \bowtie y$. Thus, since $M \sqcup D \downarrow$, $\theta(x) \sqcup \theta(y) \downarrow$. Therefore, according to Proposition 6.4, $\theta(\widehat{x}) \sqcup \theta(\widehat{y}) \downarrow$.

<u>Case A.ii.</u> $[\widehat{x}]^{-1} \subseteq VariableCut(M, D)$. According to Proposition 6.3, $\exists x \in [\widehat{x}]^{-1}, \exists y \in [\widehat{y}]^{-1}$ such that $x \bowtie y$. But Proposition 4.1 states that

$\theta(\widehat{x}) \sqsupseteq \theta(x)$ and $\theta(\widehat{y}) \sqsupseteq \theta(y)$. Therefore, according to Condition 2 of Definition 6.3, $\theta(\widehat{x}) \sqcup \theta(\widehat{y}) \downarrow$.

Case A.iii. $\widehat{x} \in DynamicCut(\widehat{M}, D)$. According to Proposition 6.3, $\exists x \in [\widehat{x}]^{-1}, \exists y \in [\widehat{y}]^{-1}$ such that $x \bowtie y$. But Proposition 4.1 states that $\theta(\widehat{y}) \sqsupseteq \theta(y)$. Therefore, according to Condition 2 of Definition 6.6, $\theta(\widehat{x}) \sqcup \theta(\widehat{y}) \downarrow$.

Case A.iv. According to Proposition 6.3, $\exists x \in [\widehat{x}]^{-1}, \exists y \in [\widehat{y}]^{-1}$ such that $x \bowtie y$. Since $\widehat{M}^{\bowtie_D}_{\bowtie} \sqcup \widehat{D} \downarrow$ and based on Proposition 2.1 ($\bowtie = \blacktriangleright\!\blacktriangleleft$), $\theta(\widehat{x}) \sqcup \theta(\widehat{y}) \downarrow$.

Case B. It will be shown that $\exists t \in Type$ such that $\forall \widehat{y} \in [\widehat{z}]_{\bowtie} \cap \widehat{D}$ and for $\{\widehat{x}\} = [\widehat{z}]_{\bowtie} \cap \widehat{M}$, $t \sqsupseteq \theta(\widehat{y})$ and $t \sqsupseteq \theta(\widehat{x})$.

Case B.i. According to Corollary 6.1, this case is not possible.

Case B.ii. $\forall \widehat{y} \in [\widehat{z}]_{\bowtie} \cap \widehat{D}$, $\widehat{y} \bowtie \widehat{x}$, and according to Proposition 6.3, $\exists y \in [\widehat{y}]^{-1}, \exists x \in [\widehat{x}]^{-1}$ such that $y \bowtie x$. Thus, according to Condition 2 of Definition 6.3, $\forall s \sqsupseteq \theta(y), \forall t \sqsupseteq \theta(x)$, $s \sqcup t \downarrow$. But according to Proposition 4.1, $\theta(\widehat{y}) \sqsupseteq \theta(y)$ and $\theta(\widehat{x}) \sqsupseteq \theta(x)$. Therefore, $\forall \widehat{y} \in [\widehat{z}]_{\bowtie} \cap \widehat{D}$, $\forall s \sqsupseteq \theta(\widehat{y})$, $\forall t \sqsupseteq \theta(\widehat{x})$, $s \sqcup t \downarrow$, and hence, $\forall \widehat{y} \in [\widehat{z}]_{\bowtie} \cap \widehat{D}, \forall t \sqsupseteq \theta(\widehat{x}), t \sqcup \theta(\widehat{y}) \downarrow$. Thus, according to Proposition 6.5, $\exists t \sqsupseteq \theta(\widehat{x}), \forall \widehat{y} \in [\widehat{z}]_{\bowtie} \cap \widehat{D}$, $t \sqsupseteq \theta(\widehat{y})$.

Case B.iii. $\forall \widehat{y} \in [\widehat{z}]_{\bowtie} \cap \widehat{D}$, $\widehat{y} \bowtie \widehat{x}$, and according to Proposition 6.3, $\exists y \in [\widehat{y}]^{-1}, \exists x \in [\widehat{x}]^{-1}$ such that $y \bowtie x$. Thus, according to Condition 2 of Definition 6.6, $\forall s \sqsupseteq \theta(y), \forall t \sqsupseteq \theta(\widehat{x})$, $s \sqcup t \downarrow$. But according to Proposition 4.1, $\theta(\widehat{y}) \sqsupseteq \theta(y)$. Therefore, $\forall \widehat{y} \in [\widehat{z}]_{\bowtie} \cap \widehat{D}$, $\forall s \sqsupseteq \theta(\widehat{y})$, $\forall t \sqsupseteq \theta(\widehat{x})$, $s \sqcup t \downarrow$, and hence, $\forall \widehat{y} \in [\widehat{z}]_{\bowtie} \cap \widehat{D}, \forall t \sqsupseteq \theta(\widehat{x}), t \sqcup \theta(\widehat{y}) \downarrow$. Thus, according to Proposition 6.5, $\exists t \sqsupseteq \theta(\widehat{x}), \forall \widehat{y} \in [\widehat{z}]_{\bowtie} \cap \widehat{D}$, $t \sqsupseteq \theta(\widehat{y})$.

Case B.iv. Since $\widehat{M}^{\bowtie_D}_{\bowtie} \sqcup \widehat{D} \downarrow$, $\exists t \sqsupseteq \theta(\widehat{x})$ such that $\forall \widehat{y} \in [\widehat{z}]_{\bowtie} \cap \widehat{D}$, $t \sqsupseteq \theta(\widehat{y})$.

Case C. It will be shown that $\exists t \in Type$ such that $\forall \widehat{x} \in [\widehat{z}]_{\bowtie}$, $t \sqsupseteq \theta(\widehat{x})$.

Case C.i. According to Corollary 6.1, this case is not possible.

<u>Cases C.ii-iv.</u> Let $\{\widehat{y}\} = [\widehat{z}]_{\bowtie} \cap \widehat{D}$. The set $[\widehat{z}]_{\bowtie} \cap \widehat{M}$ can be divided into three subsets, similar to the subcases *ii-iv*: a set $S_{iv} = \{\widehat{x} \in [\widehat{z}]_{\bowtie} \cap \widehat{M} | \widehat{x} \in \widehat{M}^{\text{✂}_D}_{\text{✂}}\}$, a set $S_{iii} = \{\widehat{x} \in [\widehat{z}]_{\bowtie} \cap \widehat{M} | \widehat{x} \in \widehat{M}^{\text{✂}_D}, \widehat{x} \notin \widehat{M}^{\text{✂}_D}_{\text{✂}}$, and $\widehat{x} \in DynamicCut(\widehat{M}, D)\}$, and a set $S_{ii} = \{\widehat{x} \in [\widehat{z}]_{\bowtie} \cap \widehat{M} | \widehat{x} \in \widehat{M}, \widehat{x} \notin \widehat{M}^{\text{✂}_D}$, and $([\widehat{x}]^{-1} \cap M) \subseteq VariableCut(M, D)\}$. It will be shown that:

1) $\exists t \sqsupseteq \theta(\widehat{y})$ such that $\forall \widehat{x} \in S_{iv}$, $t \sqsupseteq \theta(\widehat{x})$,

2) $\exists t' \sqsupseteq t$ such that $\forall \widehat{x} \in S_{iii}$, $t' \sqsupseteq \theta(\widehat{x})$,

3) $\exists t'' \sqsupseteq t'$ such that $\forall \widehat{x} \in S_{ii}$, $t'' \sqsupseteq \theta(\widehat{x})$.

By proving the above claims, it is demonstrated that $\exists t''$ such that $\forall \widehat{x} \in [\widehat{z}]_{\bowtie}$, $t'' \sqsupseteq \theta(\widehat{x})$. An example of nodes in Case C is presented in Figure 6.4.

1) Since $S_{iv} \subseteq \widehat{M}^{\text{✂}_D}_{\text{✂}}$ and since $\widehat{M}^{\text{✂}_D}_{\text{✂}} \sqcup \widehat{D} \downarrow$,

$$\exists t \sqsupseteq \theta(\widehat{y}) \text{ such that } \forall \widehat{x} \in S_{iv}, t \sqsupseteq \theta(\widehat{x}). \qquad (*)$$

2) $\forall \widehat{x} \in S_{iii}$, $\widehat{x} \bowtie \widehat{y}$ and therefore, according to Proposition 6.3, $\exists x \in [\widehat{x}]^{-1}, \exists y \in [\widehat{y}]^{-1}$ such that $x \bowtie y$. Thus, according to Condition 2 of Definition 6.6 and to Proposition 4.1, $\forall s_1 \sqsupseteq \theta(\widehat{x}), \forall s_2 \sqsupseteq \theta(\widehat{y}), s_1 \sqcup s_2 \downarrow$. More than this, since $t \sqsupseteq \theta(\widehat{y})$ (for the type t from (*)), $\forall s_1 \sqsupseteq \theta(\widehat{x}), \forall s'_2 \sqsupseteq t, s_1 \sqcup s'_2 \downarrow$, and hence, $\forall s'_2 \sqsupseteq t, s'_2 \sqcup \theta(\widehat{x}) \downarrow$. Therefore, according to Proposition 6.5 and to (*),

$$\exists t' \sqsupseteq t \sqsupseteq \theta(\widehat{y}) \text{ such that } \forall \widehat{x} \in S_{iii}, t' \sqsupseteq \theta(\widehat{x}). \qquad (**)$$

3) $\forall \widehat{x} \in S_{ii}$, $\widehat{x} \bowtie \widehat{y}$ and therefore, according to Proposition 6.3, $\exists x \in [\widehat{x}]^{-1}, \exists y \in [\widehat{y}]^{-1}$ such that $x \bowtie y$. Thus, since $x \in$

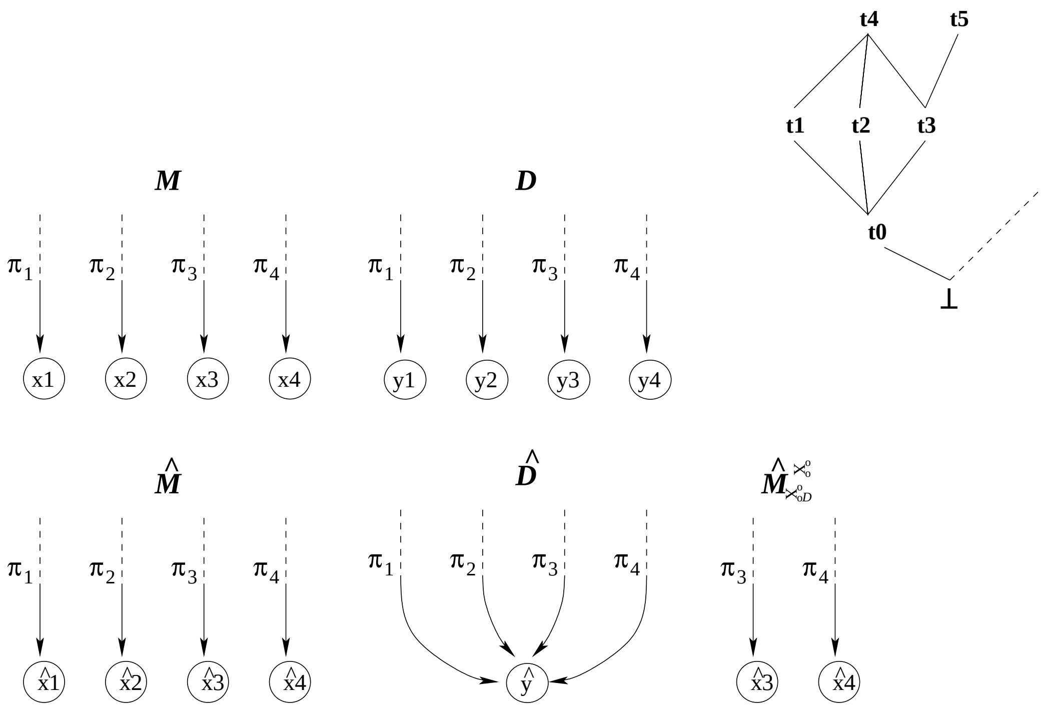


Figure 6.4: An example of nodes in Case C of the proof for Proposition 6.6. Given $\theta(x_1) = t_1, \theta(x_2) = t_2, \theta(x_3) = t_3, \theta(x_4) = t_3, \theta(y_1) = t_1, \theta(y_2) = t_1, \theta(y_2) = t_1, \theta(y_3) = t_2, \theta(y_4) = t_3, \theta(\widehat{x_1}) = t_4, \theta(\widehat{x_2}) = t_2, \theta(\widehat{x_3}) = t_3, \theta(\widehat{x_4}) = t_4, \theta(\widehat{y}) = t4$, $\widehat{M_{\ltimes}^{\rtimes D}} \sqcup \widehat{D} \downarrow$, and also $\widehat{M} \sqcup \widehat{D} \downarrow$.

$VariableCut(M, D)$, Condition 2 of Definition 6.3 holds, and therefore, according to Proposition 4.1, $\forall s_1 \sqsupseteq \theta(\widehat{x}), \forall s_2 \sqsupseteq \theta(\widehat{y}), s_1 \sqcup s_2 \downarrow$. More than this, since $t' \sqsupseteq \theta(\widehat{y})$ (for the type t' from (**)), $\forall s_1 \sqsupseteq \theta(\widehat{x}), \forall s_2' \sqsupseteq t', s_1 \sqcup s_2' \downarrow$, and hence, $\forall s_2' \sqsupseteq t', s_2' \sqcup \theta(\widehat{x}) \downarrow$. Therefore, according to Proposition 6.5 and to (**),

$$\exists t'' \sqsupseteq t' \sqsupseteq t \sqsupseteq \theta(\widehat{y}) \text{ such that } \forall \widehat{x} \in S_{ii}, t'' \sqsupseteq \theta(\widehat{x}).$$

Case D. According to Corollary 6.1, **Case D.i** is not possible, while

Cases D.ii. – D.iv. are a generalization of cases B.ii-iv and C.ii-iv.

Case E. If $|[\widehat{z}]_{\bowtie} \cap \widehat{M}| = 0$, then $[\widehat{z}]_{\bowtie} \subseteq \widehat{D}$. Since $\widehat{D}$ exists, $\exists t \in Type$ such that $\forall \widehat{y} \in [\widehat{z}]_{\bowtie}$, $t \sqsupseteq \theta(\widehat{y})$.

If $|[\widehat{z}]_{\bowtie} \cap \widehat{D}| = 0$, then $[\widehat{z}]_{\bowtie} \subseteq \widehat{M}$ and $\forall \widehat{x} \in [\widehat{z}]_{\bowtie}$, $\forall \widehat{y} \in \widehat{D}$, $\neg(\widehat{x} \bowtie \widehat{y})$. Therefore, since $\widehat{M}$ exists, $\exists t \in Type$ such that $\forall \widehat{x} \in [\widehat{z}]_{\bowtie}$, $t \sqsupseteq \theta(\widehat{x})$.

Part 2 ($\widehat{M^{\text{✂}D}_{\text{✂}}} \sqcup \widehat{D} \uparrow \Rightarrow \widehat{M} \sqcup \widehat{D} \uparrow$.) If $\widehat{M^{\text{✂}D}_{\text{✂}}} \sqcup \widehat{D} \uparrow$, then $\exists \widehat{z} \in \widehat{M^{\text{✂}D}_{\text{✂}}} \cup \widehat{D}$ such that $\neg\exists t \in Type$ for which $\forall \widehat{x} \in [\widehat{z}]_{\bowtie}, t \sqsupseteq \theta(\widehat{x})$. Since $\widehat{M^{\text{✂}D}_{\text{✂}}} \subseteq \widehat{M}$, $\exists \widehat{z} \in \widehat{M} \cup \widehat{D}$ such that $\neg\exists t \in Type$ for which $\forall \widehat{x} \in [\widehat{z}]_{\bowtie}, t \sqsupseteq \theta(\widehat{x})$, and therefore, $\widehat{M} \sqcup \widehat{D} \uparrow$.

$\square$

6.2.2 Building the Path Index

The indexing scheme used here is built on the same principles as the one described in Section 6.1, based on mother-daughter matching. The main difference is the content of the indexing keys, which now include a third element. This set is used in a two-layer indexing method.

In path indexing, each mother M has its indexing scheme defined as: $\mathcal{L}(M) = \{(R_i, D_j, V_{i,j})\}$. The pair (R_i, D_j) is the *positional index key* and represents the position D_j in the rule R_i of a matching daughter, while $V_{i,j}$ is a vector containing type values extracted from M. These vectors will be referred to as *path index vectors*. For each pair of mother-daughter that can unify, a different set of types is extracted. When M is inserted in the chart as an edge, it is placed in the entry associated with (R_i, D_j) as in the positional indexing method, but its vector $V_{i,j}$ accompanies M in the chart.

The positional index uses the daughter's position as the index key, without any need for a function to compute the key during run-time. Path indexing, however, uses a two-layer method. The first layer consists of the positional key for daughters. The second layer uses types extracted from the typed feature structure. For this, each daughter is now associated with more than one index key. The set of a daughter's index keys is: $\mathcal{L}(D) = \{(R_i, D_j, V_{i,j})\}$, where R_i is the rule number of a matching mother, D_j is D's position, and $V_{i,j}$ is the path index vector containing types extracted from D. A daughter D has a different vector defined for each mother M that can unify with D.

6.2.3 Key Extraction in Path Indexing

The vectors used as the second layer index should be of the same size for each pair of matching mothers and daughters. More than that, a vector V_i from the indexing scheme of a daughter D that matches a mother M and a vector V_j from the indexing scheme of M associated with D should refer to sets of equivalent nodes (through $\bowtie$) in the mother and in the daughter.

The types extracted for the indexing vectors are those of nodes found at the end of the *indexing paths*. An indexing path π for a mother M that unifies with a daughter D is defined as:

Definition 6.8. *If* $\widehat{M}_{\bowtie}^{\bowtie D} = \langle Q', \overline{Q'}, \theta, \delta \rangle$ *is the Dynamically Cut typed feature structure of a mother* $M = \langle Q, \overline{q}, \theta, \delta \rangle$ *with respect to a daughter* D *for which* $M \sqcup D \downarrow$ *before parsing, then* $\pi \in Path$ *is an* indexing path *iff* $\delta(\pi, \overline{q}) \in [\overline{Q'}]^{-1}$.

In the current implementation, a more efficient definition of indexing paths is used:

Definition 6.9. *If* $\widehat{M}^{\propto_D} = \langle Q', \overline{Q'}, \theta, \delta \rangle$ *is the Statically Cut typed feature structure of a mother* $M = \langle Q, \overline{q}, \theta, \delta \rangle$ *with respect to a daughter* D *for which* $M \sqcup D \downarrow$ *before parsing, then* $\pi \in Path$ *is an* indexing path *iff* $\delta(\pi, \overline{q}) \in [\overline{Q'}]^{-1}$.

The motivation for not using the *DynamicCut* is the high cost of evaluating the Condition 2 of Definition 6.6, a condition that is tested during parsing (after each edge is created). Any significant operation performed during run-time negatively affects the efficiency of the parser. An experiment using the *DynamicCut* to determine the indexing paths yielded parsing times of more than 100% slower than those of the baseline. For the *StaticCut* however, the conditions are tested off-line, with no penalty placed on parsing times.

From an implementation point of view, in order to avoid traversing the indexing path during run-time, the path index key contains pointers to the types at the ends of the indexing paths of the TFSs in the chart.

6.2.4 Using the Path Index

Inserting and retrieving edges from the chart using path indexing is similar to the general method presented in Section 6.1.2. The first layer of the index is used to insert a mother as an edge into appropriate chart entries, according to the positional keys for the daughters it can match. Along with the mother, its path index key is inserted into the chart.

When searching for a matching edge for a daughter, the search is restricted by the first indexing layer to a single entry in the chart (labeled with the positional index key for the daughter). The second layer restricts searches to the edges that have a compatible path index vector. The compatibility is defined as type unification: the type pointed to by the element $V_{i,j}(n)$ of an edge's vector $V_{i,j}$ should unify with the type pointed to by the element $V_{i,j}(n)$ of the path index vector $V_{i,j}$ of the daughter on position D_j in a rule R_i. Therefore, the unification between a mother and a daughter is attempted only when both the positional index keys and the path index vectors are compatible.

Chapter 7

Experimental Evaluation

In this chapter, a preliminary evaluation of the indexing methods presented in Chapter 6 (positional indexing and path indexing) is conducted. Details about the grammar used, the Prolog implementation, and the experimental context are given.

7.1 Resources

A pre-release version of the MERGE grammar was used for evaluating the performance of indexing. MERGE is the adaptation of the English Resource Grammar [CSLI, 2002] for TRALE [Meurers and Penn, 2002] (a HPSG parsing system built on top of ALE). This preliminary grammar has 13 rules with 2 daughters each, 4 unary rules, and 136 lexical entries. The type hierarchy contains 1157 types, with 144 introduced features.

For performance measurements, a test set containing 1970 sentences of lengths between 2 and 16 words[1] was used. This corpus was automatically generated from a smaller "seed" corpus of 196 sentences. The seed sentences consist of 98 sentences included as test sentences in this release of MERGE, 86 sentences extracted from the Wall Street Journal (Penn Tree Bank v. 2) annotated parse trees, and 12 hand-built sentences. The sentences in the test corpus were generated by replacing nouns in the seed sentences with noun phrases or with words that are both noun and verb and by using the seed sentences as subordinated clauses.

Two versions of MERGE were employed during the experimental evaluation. The first version uses a Prolog encoding for typed feature structures that does not support the representation of type constraints, while the second version uses an extended encoding of Prolog terms that allows for the representation of type constraints. Since many unification failures are caused by the type constraints, the parsing times for the second version are faster than for the first one, as it will be seen in the following sections. Also, the lack of constraints in the first version causes overgeneration, therefore more sentences are recognized as grammatically correct by the non-constrained version (only 1112 out of the initial 1970 sentences are considered correct by the constrained grammar).

The motivation for evaluating both an unconstrained and a constrained version of the same grammar resides in the need for presenting the performance of the proposed indexing methods in two extreme cases. The unconstrained grammar serves as a larger scale grammar (where more edges

[1]The coverage of this version of the MERGE grammar is quite limited, therefore the test sentences are rather short.

are added to the chart and more unifications take place). The constrained version is a more typical TFSG. The performance of the indexing methods on a real large-scale TFSG is expected to be in between these two extremes.

7.2 Prolog Data Structure

Before presenting (in the following section) the experimental evaluation of the proposed indexing methods, the data structure used to encode TFSs in Prolog is introduced. It should be mentioned that the experiments are highly dependent on the chosen data structure, which is unrelated to the proposed indexing method. Therefore, details about the TFS encoding are given here, rather than in earlier chapters.

The feature structures are encoded as Prolog terms. From the existing methods that efficiently encode TFSs ([Mellish, 1988], [Gerdemann, 1995], [Penn, 1999a]), an optimized encoding similar to the one presented in [Penn, 1999a] was chosen. As is shown in the aforementioned paper, if the feature graph is N-colourable (but not $(N-1)$-colourable), the least number of argument positions in a flat encoding is N. The feature graph is an undirected graph where each vertex represents an introduced feature, and each edge, a pair of features that are appropriate to a common type [Penn, 1999a]. Each position N in a term represents a feature with colour N in the feature graph. The encoding introduced in this thesis extends the encoding from [Penn, 1999a] by allowing for the enforcement of type constraints during unification of typed feature structures.

Figure 7.1 presents the structure used to encode TFSs as Prolog terms.

The variable `Type` encodes the type of the feature structure, while each variable $\mathtt{F}_i$ represents a feature value assigned to position i (coloured as i in the feature graph). The arity of `feats` is constant, and is the minimum number of colours N. Each of `feats`'s arguments is either a singleton variable (meaning that particular feature value is not defined for the current type or is $\perp$), a shared variable (representing structure sharing), or an embedded `tfs` term. It is possible for a `tfs` term to have a variable as its second argument (for structure sharing).

Typed feature structures: `tfs(Type,Feats,TypeRef)`.

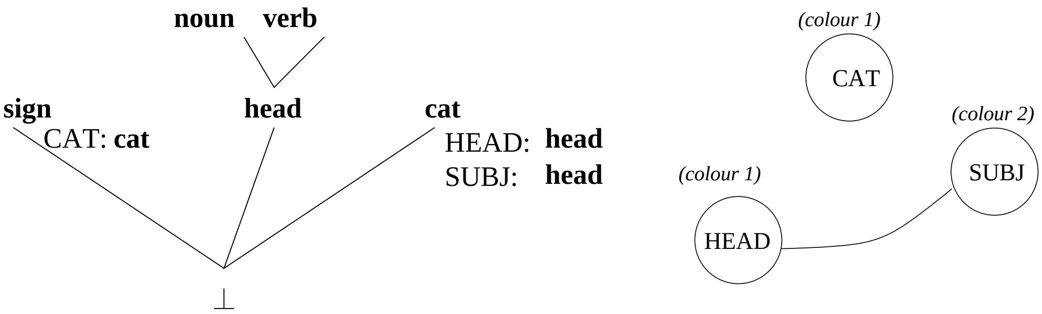
Features: `Feats=feats(`$\mathtt{F}_1$`,...,`$\mathtt{F}_N$`)`.

Types (as attribute of `Type`): `type(TypeValue,TypeRef,FeatRef)`.

Figure 7.1: The encoding of TFSs as Prolog terms

Types were encoded using the attributed variables library in SICStus Prolog [SICS, 2003]. While existing systems (such as ALE[Carpenter and Penn, 2001]) use a reference-based encoding for types, attributed variables are more suitable for indexing. Performing de-referencing each time a type is needed for the indexing key during parsing would result in slower parsing time. Attributed variables offer direct access to the encoded type.

In the Prolog encoding of a TFS (`tfs(Type,Feats,TypeRef)`) shown in Figure 7.1, the variable `Type` carries the term `type(TypeValue,TypeRef,FeatRef)` as an attribute. `TypeValue` represents the actual type value, while `FeatRef` is a pointer to the features arguments `Feats` in the TFS encoding `tfs/3`. `TypeRef` is a trigger variable (it has no value associated with it) used to enforce the type constraints. It is pointed to by the `TypeRef` variable from the `tfs/3` term. Its purpose is to delay the activation of the constraint enforcing mechanism until the unification of all features in `Feats` is completed (considering that typical Prolog systems perform the unification of arguments in a term from left to right).



$$mary \longrightarrow \begin{bmatrix} \textbf{sign} \\ \text{CAT:} \begin{bmatrix} \textbf{cat} \\ \text{HEAD: } \textbf{noun} \\ \text{SUBJ: } \textbf{head} \end{bmatrix} \end{bmatrix}$$

TFS for *mary*: `tfs(Type1,Feats1,TypeRef1).`

`Type1` attribute: `type(sign,TypeRef1,Feats1).`

`Feats1=feats(tfs(Type2,Feats2,TypeRef2),_).`

`Type2` attribute: `type(cat,TypeRef2,Feats2).`

`Feats2=feats(tfs(Type3,Feats3,TypeRef3),tfs(Type4,Feats4,TypeRef4)).`

`Type3` attribute: `type(noun,TypeRef3,Feats3).`

`Type4` attribute: `type(head,TypeRef4,Feats4).`

`Feats3=_`, `Feats4=_`.

Figure 7.2: The encoding of the TFS for the word *mary* as Prolog terms

Figure 7.2 presents the Prolog encoding for a typed feature structure representing the lexical entry for *mary*, given a type hierarchy and its coloured feature graph.

7.3 Experiments

In this section the preliminary experimental evaluations of the indexing methods proposed in Chapter 6 are presented. All experiments were carried out using the MERGE grammar described in Section 7.1. The performance was timed on a Sun Workstation with 1024 MB of memory, running an UltraSparc v.9 processor at 440 MHz with 2MB of cache. The parser was implemented in SICStus 3.10.1 for Solaris 8.

The baseline used in these experiments was the EFD parser. As shown in [Penn and Munteanu, 2003], a parser that reduces the copying of edges is already faster than several other available parsers. Also, the chosen Prolog encoding of TFSs proved to be very efficient when used in the EFD parser. All experiments were carried out using both the unconstrained and constrained versions of the MERGE grammar.

7.3.1 Evaluation using the unconstrained MERGE

For the unconstrained version (Figure 7.3), positional indexing outperformed the non-indexed parser by an average of 21% (the best improvement being 44%). Although the difference at every sentence between the parsing times for path and positional indexing is rather variable (while the difference between parsing times for positional indexing and for non-indexed parser is almost constant), the improvements in parsing times for path indexing

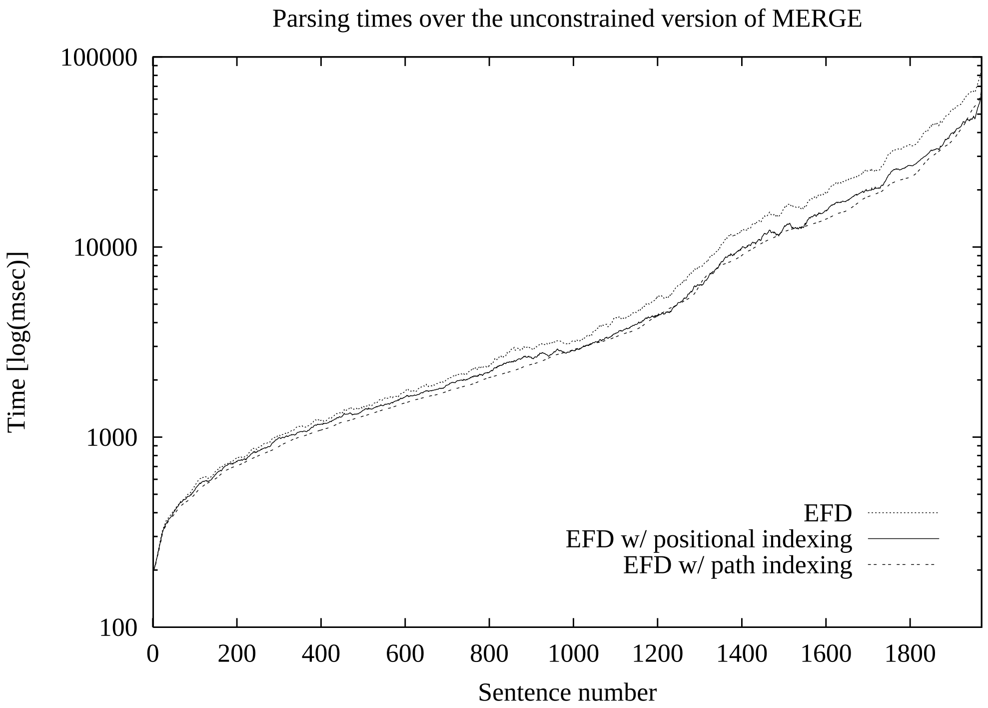


Figure 7.3: Parsing times for EFD, EFD with positional indexing, and EFD with path indexing applied to the unconstrained MERGE grammar. Sentences were numbered according to the ascending order of parsing times for path indexing.

over the non-indexed parser are better than those of positional indexing: an average of 25%, with a maximum of 45%. However, there were no significant variations in the improvements brought by both indexing method as the parsing times increased.

When using the parsing times of the positional indexing parser as a baseline, path indexing performs better than positional indexing with an average of 5% (with a maximum of 24%). These improvements, as well as both indexing methods' improvements over the non-indexed parser, are explained by a reduction in the number of unsuccessful unifications. Some examples illustrating this reduction are presented in Table 7.1.

Sentence number	Successful unifications	Failed unifications		
		EFD	EFD with positional index	EFD with path index
101	29	217	143	119
851	95	643	384	282
1357	355	2983	2092	1594
1816	557	8626	6475	4676

Table 7.1: The number of successful and failed unifications for the non-indexed and indexed parsers over the unconstrained MERGE grammar, for selected sentences where significant improvements (at least 10%) in parsing times were recorded both between positional indexing and EFD and between path indexing and positional indexing. The sentence numbers are the same as those used in Figure 7.3.

7.3.2 Evaluation using the constrained MERGE

For the constrained version (Figure 7.4), positional indexing outperformed the non-indexed parser with an average of 5% (the best improvement being 60%). Path indexing brought no improvements relative to positional indexing (showing the same average improvements as positional indexing, while the best improvement being slightly better, 62%). Similarly with the

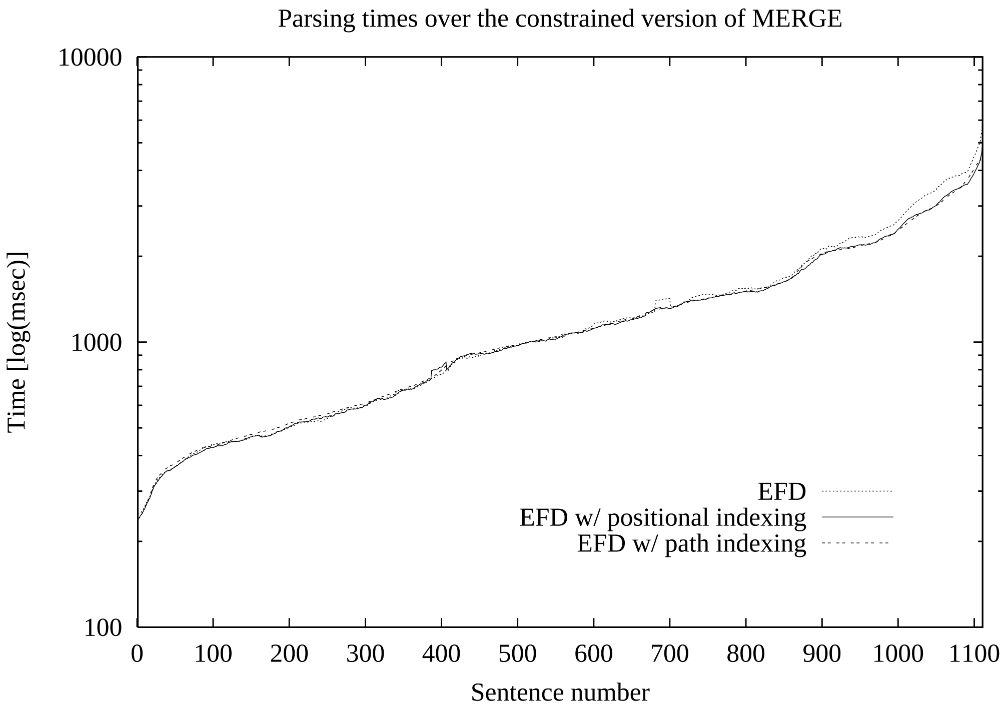


Figure 7.4: Parsing times for EFD, EFD with positional indexing, and EFD with path indexing applied to the constrained MERGE grammar. Sentences were numbered according to the ascending order of parsing times for path indexing.

unconstrained MERGE, both indexing methods exhibited a rather uniform improvement as the parsing times increased.

As can be seen in the examples[2] presented in Table 7.2, although the indexing methods avoid a significant number of failed unifications, the total number of unifications is roughly 10 times smaller than for the unconstrained version. This causes not only faster parsing times, but the indexed parser to

[2]This example is independent of the one presented in Table 7.1.

pay a significantly higher price for managing the index.

Sentence number	Successful unifications	Failed unifications		
		EFD	EFD with positional index	EFD with path index
414	18	169	143	103
681	28	127	120	103
1056	63	561	479	351

Table 7.2: The number of successful and failed unifications for the non-indexed and indexed parsers over the constrained MERGE grammar, for selected sentences where significant improvements (at least 3%) in parsing times were recorded both between positional indexing and EFD and between path indexing and positional indexing. The sentence numbers are the same as those used in Figure 7.4.

7.3.3 Comparison between statistical and non-statistical optimizations

It is not the purpose of this thesis to demonstrate the superiority of non-statistical optimizations over statistical ones. Non-statistical optimizations can be seen as a first step toward a highly efficient parser, while statistical optimization can be applied as a second step. Future work will investigate combinations of statistical and non-statistical methods to improve parsing times.

One of the purposes of non-statistical indexing is to alleviate the burden of

	Positional Indexing	Path Indexing	Quick Check
Compiling type unification	5’6”		
Compiling grammar rules	25”		
Compiling positional index	2”	2”	-
Compiling static cut	-	1’6”	-
Compiling indexing paths	-	6’45”	-
Run 300-sentence training	-	-	42’41”
Total set-up time	5’33”	13’24”	48’12”

Table 7.3: The set-up times for non-statistically indexed parsers and statistically optimized parsers for MERGE gramamr.

training a statistically optimized parser by offering comparable improvements in parsing times with significantly lower set-up times. Therefore, a quick-check parser was also built and evaluated. A comparison between the set-up times for the indexed parsers and the quick-check parser was conducted in order to better reflect the advantage of non-statistical optimizations. The set-up times are presented in Table 7.3. For quick-check, training times are computed for a 300-sentence subset of the test corpus (as prescribed in [Malouf *et al.*, 2000]). The average parse time was 8.53 seconds per sentence, roughly the same average as for the entire test corpus.

The advantage of faster set-up times for positional and path indexing can be overshadowed by slower parsing times. However, as can be seen[3]

[3]In order to maintain the clarity of these graphs, quick-check parsing times are reported separately than positional and path indexing parsing times.

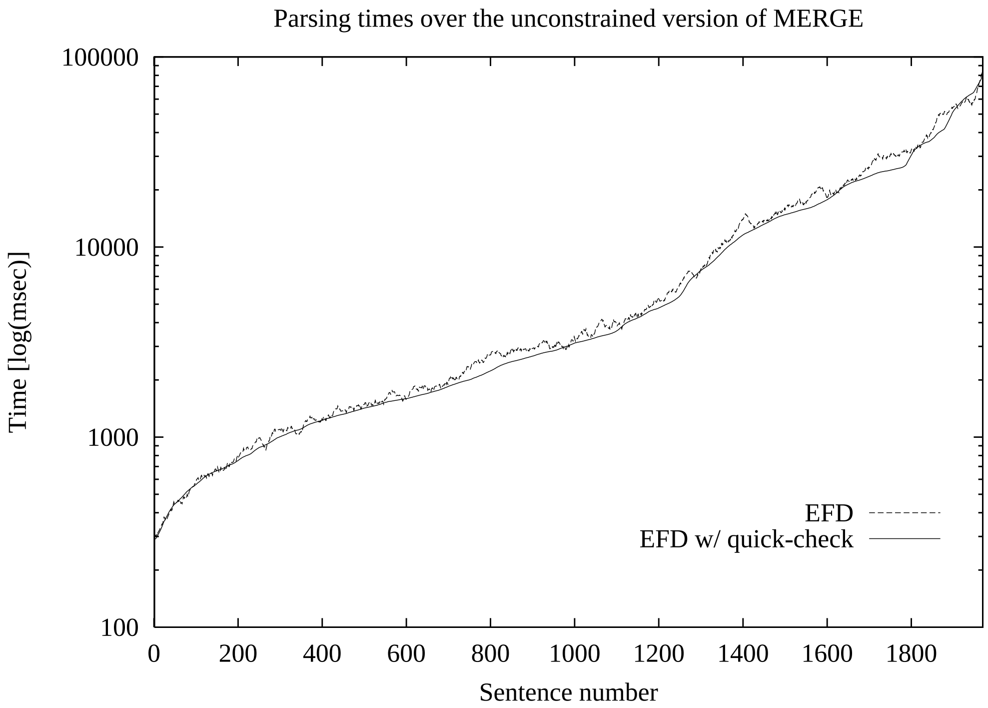


Figure 7.5: Parsing times for EFD and EFD with quick-check applied to the unconstrained MERGE grammar. The sentence numbers are the same as those used in Figure 7.3.

from Figure 7.5 and Figure 7.6, the average improvements brought by quick-check are less than (for the unconstrained MERGE) or comparable to (for the constrained MERGE) those of non-statistical indexing. For the unconstrained MERGE, quick-check improved the non-indexed EFD parsing times by only 6% (with a maximum of 42%), a value significantly lower than for the positional or path indexing. For the constrained MERGE, the average improvements of quick-check are 6% (with a maximum of 39%), which is with only 1% better than positional or path indexing. It should

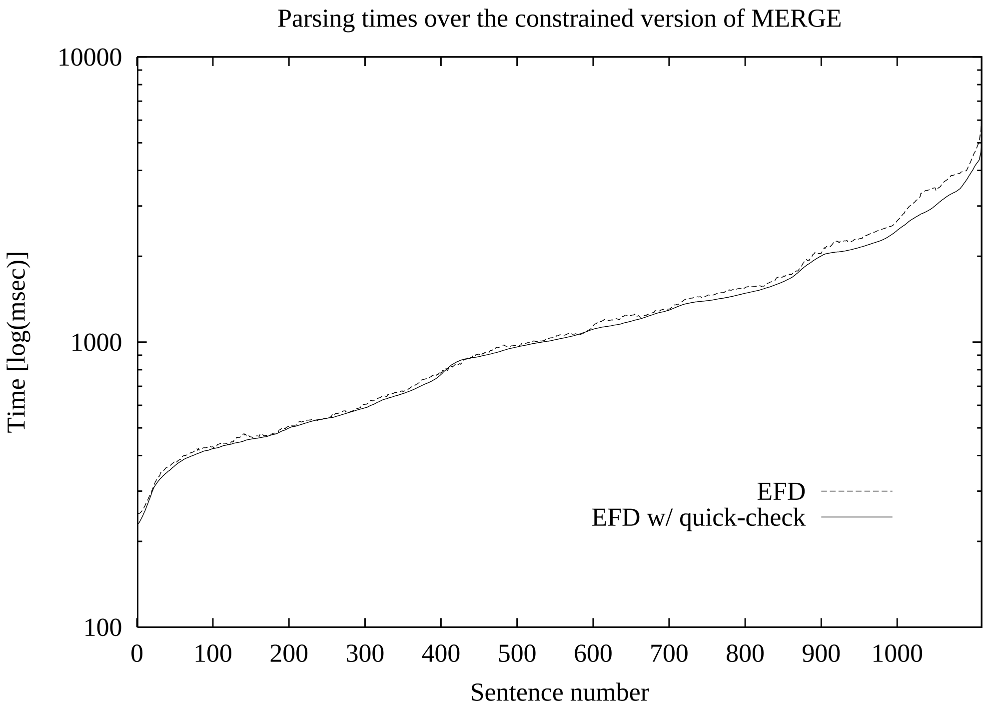


Figure 7.6: Parsing times for EFD and EFD with quick-check applied to the constrained MERGE grammar. The sentence numbers are the same as those used in Figure 7.4.

be mentioned that the quick-check evaluation presented in [Malouf *et al.*, 2000] uses only sentences with a length of at most 10 words, and the authors report only a mean figure for the parsing times (while not reporting any values for the set-up times). Also, the aforementioned paper does not specify if the training sentences are included in the test corpus; in the quick-check evaluation presented in this section, the test corpus contains the training sentences, thus offering an advantage to quick-check.

Although the number of failed unifications is smaller for the quick-check parser than for the path indexing parser (as shown in Table 7.4), the average parsing times are faster for path indexing than for quick-check in the unconstrained case. This is explained by the lower costs of managing the path index: verifying the unification of path index vectors is faster than verifying the unification of quick-check vectors. The cause for this difference resides in the static nature of non-statistical indexing: the path index vector is statically determined off-line or at compile time, and therefore the nodes at the end of the indexing paths are reachable before parsing starts, saving the time spent at run-time to fill the values in a template each time an edge is added to the chart or when a daughter is searched for in the chart. With quick-check, this is not the case – although the paths are computed off-line, many of them may not even be defined for a given feature structure.

Sentence number	Successful unifications	Failed unifications		
		EFD	EFD with path index	EFD with quick-check
101	29	217	119	26
851	95	643	282	169
1357	355	2983	1594	748
1816	557	8626	4676	1821

Table 7.4: The number of successful and failed unifications for the non-indexed, path-indexed, and quick-check parsers over the unconstrained MERGE grammar. The sentence numbers are the same as those used in Figure 7.3 and in Table 7.1.

7.4 Evaluation on Other UBGs

In order to demonstrate the utility of indexing for UBGs, the general indexing strategy proposed earlier in this thesis is evaluated by conducting two more experiments. The first experiment uses Alvey, an untyped grammar with smaller feature structures and more rules than MERGE. The second experiment is carried out by benchmarking a set of CFGs with no typing, but atomic categories and a very large number of rules.

7.4.1 The Alvey Grammar

The English grammar developed within the Alvey Natural Language Tools [Grover *et al.*, 1993] is a wide-coverage morphosyntactic and semantic analyzer, based on a formalism similar to that of Generalized Phrase Structure Grammar [Gazdar *et al.*, 1985]. John Carroll's Prolog port of the Alvey English grammar was used for this experimental evaluation.

The indexing strategy used for Alvey has the same general structure as that presented in Section 4.3.1: when searching for a matching edge for a daughter in the chart, only chart entries with an index key matching that daughter's index key are visited. The Prolog implementation of Alvey has feature structures encoded as terms. Since there is no typing in Alvey, and since the feature structures are small[4] (compared to MERGE), the index key associated with each daughter and each edge is the functor of the Prolog term encoding the category. Since the successful unifications of the functors

[4]For example, the lexical entry for the proper noun *kim* has 13 feature-value pairs in Alvey, while in MERGE it has 206.

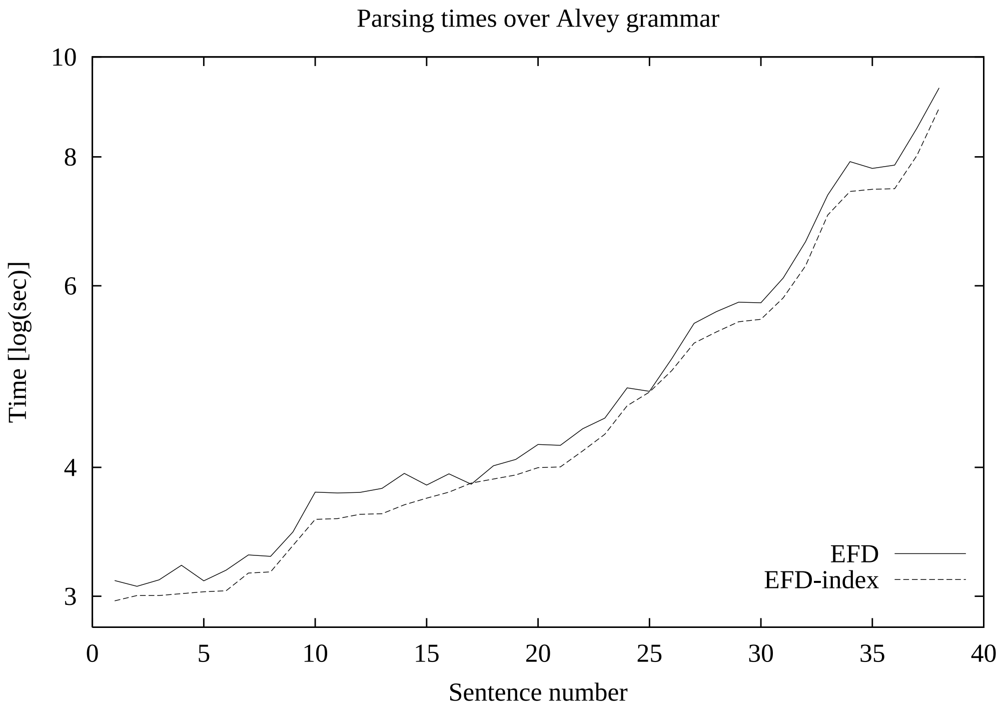


Figure 7.7: Parsing times for EFD and EFD-indexing applied to the Alvey grammar. Sentences were re-numbered and sorted according to the ascending order of parsing times for EFD-indexing. Only sentences that are parsing with EFD in more than 3 seconds are included in this figure.

do not guarantee the successful unifications of the terms, this is far from a "perfect" indexing. Indeed, the improvements in parsing times are less than those recorded for the MERGE grammar. However, the cost of maintaining an indexing scheme even as simple as positional indexing would overshadow the improvements in parsing times, due to the small category size (and consequently, the small unification costs). There are 780 rules in Alvey, compared to only 17 in MERGE, leading to an unmanageable number of hash entries for positional indexing.

The evaluation set consisted of all test sentences included in the Prolog port of Alvey (182 sentences of lengths from 2 to 31 words). Even if the indexing scheme is very simple, on average the indexed parser performs 4% better than the non-indexed EFD parser, with a maximum improvement of 10% (Figure 7.7).

7.4.2 Penn Treebank CFG

CFGs are normally not considered UBGs, because they have atomic categories, and the process of unifying atomic entities is merely a simple equality check. However, as is shown in this section, indexing methods can be successfully applied to CFGs, with the results demonstrating significant improvement, especially for large-scale CFGs.

The index key for each daughter is the daughter's category itself (a category index). The indexing scheme $L(Mother)$ contains only the daughters that are guaranteed to match with a specific $Mother$ (thus creating a "perfect" index). This indexing scheme is preferred to a positional index, since it is usual for CFGs to have a large number of rules, with a significant number of daughters. As a result, a positional index would simply need too many index entries (a number equal to the total number of daughters in the grammar). Using the categories as index keys limits the number of entries to the number of different categories.

The search takes place only in the hash entry associated with that daughter's category. This increases to 100% the ratio of successful

Number of rules	Successful unifications	Failed unifications	Success rate (%)
124	104	1,766	5.56
736	2,904	189,528	1.51
3196	25,416	3,574,138	0.71
9008	195,382	56,998,866	0.34
14193	655,403	250,918,711	0.26
20999	1,863,523	847,204,674	0.21

Table 7.5: Successful unification rate for the non-indexed CFG parser.

unifications[5], representing a significant gain in terms of parsing times. Table 7.5 illustrates the significance of this gain by presenting the successful unification rate for the non-indexed CFG parser.

Fifteen CFGs with atomic categories were built from the Wall Street Journal (Penn Tree Bank v. 2) annotated parse trees, by constructing a rule from each sub-tree of every parse tree, and removing the duplicates. The grammars were extracted from the first ten directories of the Wall Street Journal collection (`00`–`09`), consisting of the files `wsj_0001.mrg` to `wsj_0999.mrg`, as shown in Table 7.6. Each of the fifteen grammars contains rules extracted from the first N files in `wsj_0001.mrg`...`wsj_0999.mrg`, with N chosen as: 5, 10, 15, 30, 50, 100, 150, 200, 250, 300, 350, 400, 500, 700, 900.

[5] 100% represents the ratio of successful unifications in the chart, without counting the failed unifications between index keys, when looking for the correct hash entry for a daughter or an edge.

Grammar no.	WSJ directories	Number of WSJ files	Lexicon size	Number of Rules
1	00	5	188	124
2	00	10	756	473
3	00	15	1167	736
4	00	30	2335	1369
5	00	50	5645	3196
6	00–01	100	8948	5246
7	00–01	129	11242	6853
8	00–02	200	13164	7984
9	00–02	250	14730	9008
10	00–03	300	17555	10834
11	00–03	350	18861	11750
12	00–04	400	20359	12696
13	00–05	481	20037	13159
14	00–07	700	27404	17682
15	00–09	901	32422	20999

Table 7.6: The grammars extracted from the Wall Street Journal directories of the PTB II.

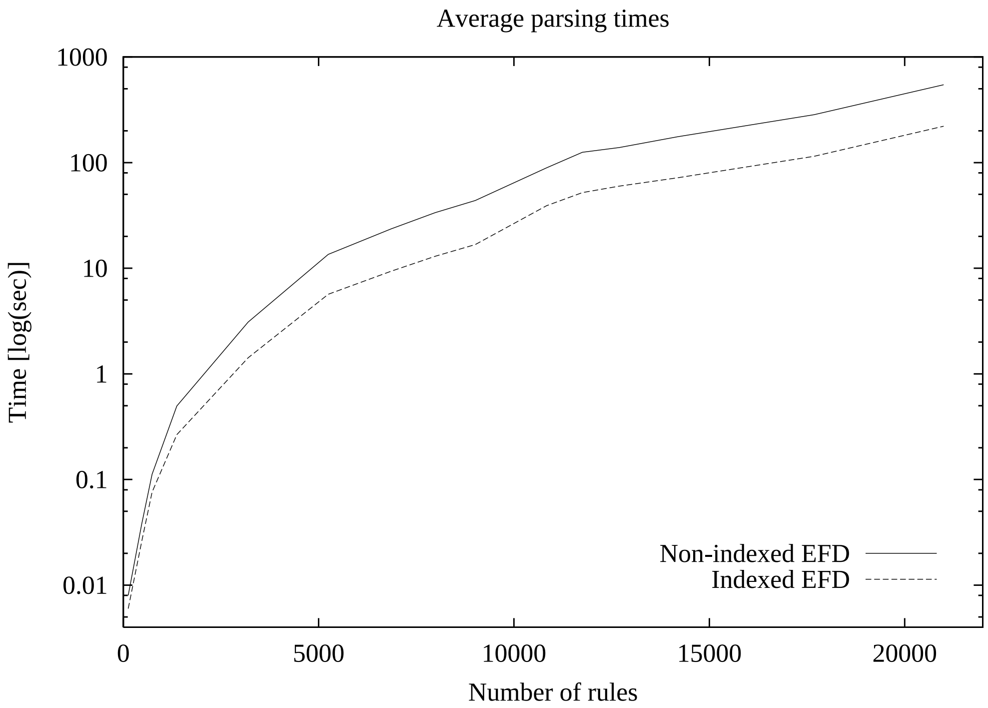


Figure 7.8: Parsing times for EFD and EFD-indexing applied to CFGs with atomic categories.

The test set contained 5 sentences of lengths between 13 and 18 words. The sentences were chosen from the first 5 files, to ensure that they will be parsed by every extracted grammar. Figure 7.8 shows that even for smaller numbers of rules, the indexed parser outperforms the non-indexed version. The improvements in parsing times for the indexed parser over those for the non-indexed parser range from a minimum of 25% to a maximum of 62%. All grammars with more than 3000 rules (11 out of the total 15 grammars) exhibit improvements in parsing times of more than 50%, while only the smallest grammar shows improvements under 30%. Although "unification" costs are small for atomic CFGs, using an indexing method is well justified. The difference in improvements from Penn Tree Bank CFG to MERGE TFSG is explained by the large branching factor of the CFG rules, by the "perfect" index, and by the sheer size of the charts in number of edges. All of these, as mentioned in Section 5.3, are important factors in an indexing method's success.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

This thesis presented an investigation of possible improvements in parsing times for typed feature structure grammars by means of indexing. A review of the existing research on improving TFSG parsing times revealed a lack of an integrated approach to indexing, as well as of a thorough analysis of the grammar rules in TFS-based parsers that can lead to the development of more efficient parsers. A theoretical framework to support a static analysis of grammar rules was set up, and an indexing strategy based on the static analysis was proposed.

The indexing method proposed here is suitable for several classes of unification-based grammars. The index keys are determined statically and are based on an *a priori* analysis of grammar rules. A major advantage of such indexing methods is the elimination of the lengthy training processes needed by statistical methods. Although a non-statistical method, indexing through the static analysis of grammar rules can be combined with methods based on statistical evidence of mother-daughter unifications.

The preliminary experimental evaluation carried out over several unification-based grammars demonstrates that indexing through static analysis is a promising optimization for typed feature structure grammars. The improvements in parsing time are comparable to those of statistically optimized parsers, while their set-up time is significantly lower.

8.2 Future Work

Several research directions using this thesis as a starting point can be identified:

Indexing Constraints. The current static analysis of grammar rules used as a basis for indexing does not consider type constraints. Future work will investigate an extension of the indexing method in order to accommodate the use of type constraints, with the potential to significantly reduce parsing times.

The Static and Dynamic Cut Definitions. The definitions for the Static and the Dynamic Cuts are not guaranteed to be maximally specific (although they are probably close), thus it is possible that certain nodes are not included in the Static or Dynamic Cut, and yet can be discarded. The inclusion of such nodes does not affect the formal properties of the Cuts, but it might diminish the improvements brought about by indexing. Future research will investigate various extensions of the Cuts, in order to include all possible nodes.

The Static and Dynamic Cut Implementation. The theoretical benefits of the Static Cut are exploited in the path indexing method. A possible use of the Static and Dynamic Cut that will be investigated in the future is the replacement of the normal unification between the most general satisfiers of a mother and a daughter with the unification of their Statically Cut most general satisfiers.

Index Implementation. In this thesis, the indexing is implemented as a hash, a structure that is not always efficient. As mentioned in Section 5.3, several indexing structures (such as B-trees) commonly used in databases are not suitable for parsing. Future research will thoroughly analyze other structures used in databases, and identify the most appropriate one for indexed parsing.

Indexing Strategies. While the tree-based indexing (overviewed in Section 5.2.3) has poor performance when the number of insertions is quite large, its theoretical benefits cannot be ignored. A possible adaptation of this technique to chart parsing will be studied.

Feature Encoding. The current method for encoding feature structures proved its efficiency. However, further improvements will be sought that will allow for the representation of different extensions to typed feature structures, such as inequations.

Statistical Improvements. The purpose of this thesis was not to discredit existing statistical methods. Future work will include the investigation of possible improvements through an integration of statistical methods, such as feature re-ordering or improved quick check tests.

Experimental Evaluation. At the time of writing this thesis, the only wide-coverage typed feature structure grammar available to the author was the preliminary version of MERGE. In order to thoroughly evaluate the indexing based on static analysis, a much larger experimental set (a larger grammar and a larger test corpus containing no artificially generated sentences) will be sought in the future.

Bibliography

[Allen, 1994] J. Allen. *Natural Language Understanding.* Benjamin/Cummings Publishing, 1994.

[Belkin and Croft, 1992] N.J. Belkin and W.B. Croft. Information filtering and information retrieval: Two sides of the same coin? *Communications of the ACM*, 35(12), 1992.

[Bertino *et al.*, 1997] E. Bertino, B. C. Ooi, R. Sacks-Davis, K-L Tan, J. Zobel, B. Shidlovsky, and B. Catania. *Indexing Techniques for Advanced Database Systems.* Kluwer Academic Publishers, 1997.

[Brown and Miller, 1996] K. Brown and J. Miller, editors. *Concise Encyclopedia of Syntactic Theories.* Pergamon – Elsevier Science, 1996.

[Carpenter and Penn, 2001] B. Carpenter and G. Penn. *The Attribute Logic Engine*, 2001.

[Carpenter, 1992] B. Carpenter. *The Logic of Typed Feature Structures.* Cambridge University Press, 1992.

[Colmerauer, 1982] A. Colmerauer. Prolog and infinite trees. In K.L. Clark and S.-A. Tirnlund, editors, *Logic Prograraming*, pages 231–252. Academic Press, 1982.

[Cooper, 1996] R.P. Cooper. Head-driven phrase structure grammar. In K. Brown and J. Miller, editors, *Concise Encyclopedia of Syntactic Theories*. Pergamon – Elsevier Science, 1996.

[CSLI, 2002] CSLI. CSLI Lingo. http://lingo.stanford.edu/csli, 2002.

[Elmasri and Navathe, 2000] R. Elmasri and S. Navathe. *Fundamentals of database systems*. Addison-Wesley, 2000.

[Gazdar and Mellish, 1989] G. Gazdar and C. Mellish. *Natural Language Processing in Prolog*. Addison-Wesley, 1989.

[Gazdar *et al.*, 1985] G. Gazdar, E. Klein, G. K. Pullum, and I. A. Sag. *Generalized Phrase Structure Grammar*. Blackwell, 1985.

[Gerdemann, 1995] D. Gerdemann. Term encoding of typed feature structures. In *Proceedings of the Fourth International Workshop on Parsing Technologies*, 1995.

[Graf, 1995] P. Graf. Substitution tree indexing. In *Proceedings of the 6th International Conference on Rewriting Techniques and Applications*, 1995.

[Graf, 1996] P. Graf. *Term Indexing*. Springer, 1996.

[Grover *et al.*, 1993] C. Grover, J. Carroll, and E. Briscoe. The alvey natural language tools grammar (4th release). Technical Report 284, Computer Laboratory, Cambridge University, Cambridge, UK, 1993.

[Hammer and Chan, 1976] M. Hammer and A. Chan. Index selection in a self-adaptive database management system. In *Proceedings of the 1976 ACM SIGMOD International Conference on Management of Data*, Washington, D.C., U.S.A., 1976.

[Jurafsky and Martin, 2000] D.S. Jurafsky and J.H. Martin. *Speech and Language Processing.* Prentice Hall, 2000.

[Kay *et al.*, 1994] Martin Kay, Jean Mark Gawron, and Peter Norvig. *Verbmobil: A Translation System For Face-To-Face Dialog.* CSLI Publications, 1994.

[Kiefer *et al.*, 1999] B. Kiefer, H.U. Krieger, J. Carroll, and R. Malouf. A bag of useful techniques for efficient and robust parsing. In *Proceedings of the 37th Annual Meeting of the ACL*, 1999.

[Malouf *et al.*, 2000] R. Malouf, J. Carrol, and A. Copestake. Efficient feature structure operations without compilation. *Natural Language Engineering*, 6(1), 2000.

[Manolopoulos *et al.*, 1999] Y. Manolopoulos, Y. Theodoridis, and V. J. Tsotras. *Advanced Database Indexing.* Kluwer Academic Publishers, 1999.

[Matheson, 1997] Colin Matheson. HPSG grammars in ALE. Course Notes, Department of Linguistics, University of Edinburgh, 1997. http://www.ltg.hcrc.ed.ac.uk/projects/ledtools/ale-hpsg/.

[Mellish, 1988] C. Mellish. Implementing systemic classification by unification. *Computational Linguistics*, 14(1), 1988.

[Mellish, 1991] C. Mellish. Graph-encodable description spaces. Technical Report DYANA R3.2B, University of Edinburgh, 1991.

[Mellish, 1992] C. Mellish. Term-encodable description spaces. In D.R. Brough, editor, *Logic Programming – New Frontiers*. Kluwer Academic Publishers, 1992.

[Meurers and Penn, 2002] D. Meurers and G. Penn. *Trale Milca Environment v. 2.1.4*. http://ling.ohio-state.edu/~dm, 2002.

[Meurers, 2002] D. Meurers. Constraint-based grammar implementation. Course Notes, Department of Linguistics, Ohio State University, 2002.

[Mueck and Polaschek, 1997] T.A. Mueck and M.L. Polaschek. *Index Data Structures in Object-Oriented Databases*. Kluwer Academic Publishers, 1997.

[Ninomiya *et al.*, 2002] T. Ninomiya, T. Makino, and J. Tsujii. An indexing scheme for typed feature structures. In *Proceedings of the 19th International Conference on Computational Linguistics*, 2002.

[Oepen and Carroll, 2000] S. Oepen and J. Carroll. Parser engineering and performance profiling. *Natural Language Engineering*, 6(1), 2000.

[Penn and Munteanu, 2003] G. Penn and C. Munteanu. A tabulation-based parsing method that reduces copying. In *Proceedings of the 41st Annual Meeting of the ACL*, Sapporo, Japan, 2003.

[Penn and Popescu, 1997] G. Penn and O. Popescu. Head-driven generation and indexing in ALE. In *ACL Workshop on Computational Environments for Grammar Development and Linguistic Engineering*, 1997.

[Penn, 1999a] G. Penn. An optimised Prolog encoding of typed feature structures. In *Arbeitspapiere des SFB 340*, number 138. 1999.

[Penn, 1999b] G. Penn. Optimising don't-care non-determinism with statistical information. In *Arbeitspapiere des SFB 340*, number 140. 1999.

[Penn, 1999c] G. Penn. A parsing algorithm to reduce copying in Prolog. In *Arbeitspapiere des SFB 340*, number 137. 1999.

[Penn, 2000] G. Penn. *The Algebraic Structure of Attributed Type Signatures.* PhD thesis, School of Computer Science, Carnegie Mellon University, 2000.

[Pereira and Shieber, 1987] F.C.N. Pereira and S.M. Shieber. *Prolog and Natural Language Analysis.* CSLI Publications, 1987.

[Pollard and Sag, 1987] C. Pollard and I. Sag. *Information-Based Syntax and Semantics.* CSLI Publications, 1987.

[Pollard and Sag, 1994] C. Pollard and I. Sag. *Head-driven Phrase Structure Grammar.* The University of Chicago Press, 1994.

[Pollard, 1997] C. Pollard. Lectures on the foundations of HPSG. Course Notes, Department of Linguistics, Ohio State University, 1997.

[Ramakrishnan *et al.*, 2001] I.V. Ramakrishnan, R. Sekar, and A. Voronkov. Term indexing. In *Handbook of Automated Reasoning*, volume II, chapter 26. Elsevier Science, 2001.

[Ramesh *et al.*, 2001] G. Ramesh, W. Maniatty, and M. Zaki. Indexing and data access methods for database mining. Technical Report 01-01, Department of Computer Science, University of Albany, Albany, N.Y., U.S.A., 2001.

[Shieber, 1986] S.M. Shieber. *An Introduction to Unification-Based Approaches to Grammar*. CSLI Publications, 1986.

[SICS, 2003] SICS. SICStus Prolog. http://www.sics.se/sicstus, 2003.

[Torisawa and Tsujii, 1995] K. Torisawa and J. Tsujii. Compiling HPSG-style grammar to object-oriented language. In *roceedings of the Natural Language Processing Pacific Rim Symposium*, Seoul, Korea, 1995.

[Torisawa *et al.*, 2000] K. Torisawa, K. Nishida, Y. Miyao, and J. Tsujii. An HPSG parser with CFG filtering. *Natural Language Engineering*, 6(1), 2000.

[Uszkoreit, 1996] H. Uszkoreit. A note on the essence of hpsg and its rôle in computational linguistics. http://www.coli.uni-sb.de/~hansu/hpsg.html, 1996.

[van Noord, 1997] G. van Noord. An efficient implementation of the head-corner parser. *Computational Linguistics*, 23(3), 1997.

[Wintner, 1997] S. Wintner. *An Abstract Machine for Unification Grammars*. PhD thesis, Technion – Israel Institute for Technology, 1997.