Optimizing Typed Feature Structure Grammar Parsing through Non-Statistical Indexing

Cosmin Munteanu and Gerald Penn
University of Toronto
10 King's College Rd.
Toronto M5S 3G4
Canada
{mcosmin,gpenn}@cs.toronto.edu

Abstract

This paper introduces an indexing method based on static analysis of grammar rules and type signatures for typed feature structure grammars (TFSGs). The static analysis tries to predict at compile-time which feature paths will cause unification failure during parsing at run-time. To support the static analysis, we introduce a new classification of the instances of variables used in TFSGs, based on what type of structure sharing they create. The indexing actions that can be performed during parsing are also enumerated. Non-statistical indexing has the advantage of not requiring training, and, as the evaluation using large-scale HPSGs demonstrates, the improvements are comparable with those of statistical optimizations. Such statistical optimizations rely on data collected during training, and their performance does not always compensate for the training costs.

1 Introduction

Developing efficient all-paths parsers has been a long-standing goal of research in computational linguistics. One particular class still in need of parsing time improvements is that of TFSGs. While simpler formalisms such as context-free grammars (CFGs) also face slow all-paths parsing times when the size of the grammar increases significantly, TFSGs (which generally have fewer rules than large-scale CFGs) become slow as a result of the complex structures used to describe the grammatical categories. In HPSGs (Pollard and Sag, 1994), one category description could contain hundreds of feature values. This has been a barrier in transferring CFG-successful techniques to TFSG parsing.

For TFSG chart parsers, one of the most time-consuming operations is the retrieval of categories from the chart during rule completion (closing of constituents in the chart under a grammar rule). Looking in the chart for a matching edge for a daughter is accomplished by attempting unifications with edges stored in the chart, resulting in many failed unifications. The large and complex structure of TFS descriptions (Carpenter, 1992) leads to slow unification times, affecting the parsing times. Thus, failing unifications must be avoided during retrieval from the chart.

To our knowledge, there have been only four methods proposed for improving the retrieval component of TFSG parsing. One (Penn and Munteanu, 2003) addresses only the cost of copying large categories, and was found to reduce parsing times by an average of 25% on a large-scale TFSG (MERGE). The second, a statistical method known as quick-check (Malouf et al., 2000), determines the paths that are likely to cause unification failure by profiling a large sequence of parses over representative input, and then filters unifications at run-time by first testing these paths for type consistency. This was measured as providing up to a 50% improvement in parse times on the English Resource Grammar (Flickinger, 1999, ERG). The third (Penn, 1999b) is a similar but more conservative approach that uses the profile to re-order sister feature values in the internal data structure. This was found to improve parse times on the ALE HPSG by up to 33%.

The problem with these statistical methods is that the improvements in parsing times may not justify the time spent on profiling, particularly during grammar development. The static analysis method introduced here does not use profiling, although it does not preclude it either. Indeed, an evaluation of statistical methods would be more relevant if measured on top of an adequate extent of non-statistical optimizations. Although quick-check is thought to produce parsing time improvements, its evaluation used a parser with only a superficial static analysis of chart indexing.

That analysis, rule filtering (Kiefer et al., 1999), reduces parse times by filtering out mother-daughter unifications that can be determined to fail at compile-time. True indexing organizes the data (in this case, chart edges) to avoid unnecessary retrievals altogether, does not require the operations that it performs to be repeated once full unification is deemed necessary, and offers the support for easily adding information extracted from further static analysis of the grammar rules, while maintaining the same indexing strategy. Flexibility is one of the reasons for the successful employment of indexing in databases (Elmasri and Navathe, 2000) and automated reasoning (Ramakrishnan et al., 2001).

In this paper, we present a general scheme for indexing TFS categories during parsing (Section 3). We then present a specific method for statically analyzing TFSGs based on the type signature and the structure of category descriptions in the grammar rules, and prove its soundness and completeness (Section 4.2.1). We describe a specific indexing strategy based on this analysis (Section 4), and evaluate it on two large-scale TFSGs (Section 5). The result is a purely non-statistical method that is competitive with the improvements gained by statistical optimizations, and is still compatible with further statistical improvements.

2 TFSG Terminology

TFSs are used as formal representatives of rich grammatical categories. In this paper, the formalism from (Carpenter, 1992) will be used. A TFSG is defined relative to a fixed set of types and set of features, along with constraints, called *appropriateness conditions*. These are collectively known as the *type signature* (Figure 3). For each type, appropriateness specifies all and only the features that must have values defined in TFSs of that type. It also specifies the types of the values that those features can take. The set of types is partially ordered, and has a unique most general type ($\perp$ – "bottom"). This order is called *subsumption* ($\sqsubset$): more specific (higher) types inherit appropriate features from their more general (lower) supertypes. Two types t_1 and t_2 *unify* ($t_1 \sqcup t_2 \downarrow$) iff they have a least upper bound in the hierarchy. Besides a type signature, TFSGs contain a set of grammar (phrase) rules and lexical descriptions. A simple example of a lexical description is: $john \Longrightarrow \mathrm{SYNSEM}:(\mathrm{SYN}:np \wedge \mathrm{SEM}:j)$, while an example of a phrase rule is given in Figure 1.

$$
\begin{aligned}
&(\mathrm{SYN}:s \wedge \mathrm{SEM}:(VPSem, \mathrm{AGENT}:NPSem)) \Longrightarrow \\
&\qquad (\mathrm{SYN}:np \wedge \mathrm{AGR}:Agr \wedge \mathrm{SEM}:NPSem), \\
&\qquad (\mathrm{SYN}:vp \wedge \mathrm{AGR}:Agr \wedge \mathrm{SEM}:VPSem).
\end{aligned}
$$

Figure 1: A phrase rule stating that the syntactic category `s` can be combined from `np` and `vp` if their values for `agr` are the same. The semantics of `s` is that of the verb phrase, while the semantics of the noun phrase serves as agent.

2.1 Typed Feature Structures

A TFS (Figure 2) is like a recursively defined record in a programming language: it has a type and features with values that can be TFSs, all obeying the appropriateness conditions of the type signature. TFSs can also be seen as rooted graphs, where arcs correspond to features and nodes to substructures. A node typing function $\theta(q)$ associates a type to every node q in a TFS. Every TFS F has a unique starting or root node, q_F. For a given TFS, the feature value partial function $\delta(f,q)$ specifies the node reachable from q by feature f when one exists. The path value partial function $\delta(\pi,q)$ specifies the node reachable from q by following a path of features π when one exists. TFSs can be unified as well. The result represents the most general consistent combination of the information from two TFSs. That information includes typing (by unifying the types), feature values (by recursive unification), and structure sharing (by an equivalence closure taken over the nodes of the arguments). For large TFSs, unification is computationally expensive, since all the nodes of the two TFSs are visited. In this process, many nodes are collapsed into equivalence classes because of structure sharing. A node x in a TFS F with root q_F and a node x' in a TFS F' with root $q_{F'}$ are equivalent ($\bowtie$) with respect to $F \sqcup F'$ iff $x = q_F$ and $x = q_{F'}$, or if there is a path π such that $\delta_{F \sqcup F'}(\pi, q_F) = x$ and $\delta_{F \sqcup F'}(\pi, q_{F'}) = x'$.

$$
\begin{bmatrix}
\textbf{throwing} & \\
\mathrm{THROWER:} & \begin{bmatrix} \textbf{index} & \\ \mathrm{PERSON:} & \textbf{third} \\ \mathrm{NUMBER:} & \textbf{[1]singular} \\ \mathrm{GENDER:} & \textbf{masculine} \end{bmatrix} \\
\mathrm{THROWN:} & \begin{bmatrix} \textbf{index} & \\ \mathrm{PERSON:} & \textbf{third} \\ \mathrm{NUMBER:} & \textbf{[1]} \\ \mathrm{GENDER:} & \textbf{neuter} \end{bmatrix}
\end{bmatrix}
$$

Figure 2: A TFS. Features are written in uppercase, while types are written with bold-face lowercase. Structure sharing is indicated by numerical tags, such as **[1]**.

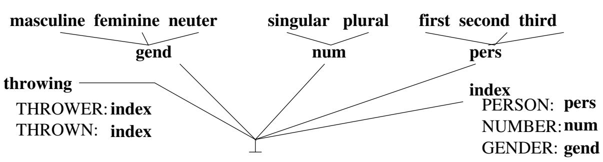

Figure 3: A type signature. For each type, appropriateness declares the features that must be defined on TFSs of that type, along with the type restrictions applying to their values.

2.2 Structure Sharing in Descriptions

TFSGs are typically specified using *descriptions*, which logically denote sets of TFSs. Descriptions can be more terse because they can assume all of the information about their TFSs that can be inferred from appropriateness. Each non-disjunctive description can be associated with a unique most general feature structure in its denotation called a *most general satisfier* (*MGSat*). While a formal presentation can be found in (Carpenter, 1992), we limit ourselves to an intuitive example: the TFS from Figure 2 is the *MGSat* of the description: $throwing \wedge \text{THROWER}: (\text{PERSON}: third \wedge \text{NUMBER}: (singular \wedge Nr), \text{GENDER}: masculine) \wedge \text{THROWN}: (\text{PERSON}: third \wedge \text{NUMBER}: Nr, \text{GENDER}: neuter)$. Descriptions can also contain variables, such as Nr.

Structure sharing is enforced in descriptions through the use of variables. In TFSGs, the scope of a variable extends beyond a single description, resulting in structure sharing between different TFSs. In phrase structure rules (Figure 1), this sharing can occur between different daughter categories in a rule, or between a mother and a daughter. Unless the term *description* is explicitly used, we will use "mother" and "daughter" to refer to the *MGSat* of a mother or daughter description.

We can classify instances of variables based on what type of structure sharing they create. *Internal variables* are the variables that represent internal structure sharing (such as in Figure 2). The occurrences of such variables are limited to a single category in a phrase structure rule. *External variables* are the variables used to share structure between categories. If a variable is used for structure sharing both inside a category and across categories, then it is also considered an external variable. For a specific category, two kinds of external variable instances can be distinguished, depending on their occurrence relative to the parsing control strategy: *active external variables* and *inactive external variables*. Active external variables are instances of external variables that are shared between the description of a category D and one or more descriptions of categories in the same rule as D visited by the parser before D as the rule is extended (completed). Inactive external variables are the external variable instances that are not active. For example, in bottom-up left-to-right parsing, all of a mother's external variable instances would be active because, being external, they also occur in one of the daughter descriptions. Similarly, all of the leftmost daughter's external variable instances would be inactive because this is the first description used by the parser. In Figure 1, Agr is an active external variable in the second daughter, but it is inactive in the first daughter.

The active external variable instances are important for path indexing (Section 4.2), because they represent the points at which the parser must copy structure between TFSs. They are therefore substructures that must be provided to a rule by the parsing chart if these unifications could potentially fail. They also represent shared nodes in the *MGSats* of a rule's category descriptions. In our definitions, we assume without loss of generality that parsing proceeds bottom-up, with left-to-right of rule daughters. This is the ALE system's (Carpenter and Penn, 1996) parsing strategy.

Definition 1. *If $D_1, \ldots, D_n$ are daughter descriptions in a rule and the rules are extended from left to right, then $Ext(MGSat(D_i))$ is the set of nodes shared between $MGSat(D_i)$ and $MGSat(D_1) \ldots MGSat(D_{i-1})$. For a mother description M, $Ext(MGSat(M))$ is the set of nodes shared with any daughter in the same rule.*

Because the completion of TFSG rules can cause the categories to change in structure (due to external variable sharing), we need some extra notation to refer to a phrase structure rule's categories at different times during a single application of that rule. By $\widehat{M}$ we symbolize the mother M after M's rule is completed (all of the rule's daughters are matched with edges in the chart). $\widehat{D}$ symbolizes the daughter D after all daughters to D's left in D's rule were unified with edges from the chart. An important relation exists between M and $\widehat{M}$: if q_M is M's root and $\widehat{q_M}$ is $\widehat{M}$'s root, then $\forall x \in M, \forall \widehat{x} \in \widehat{M}$ such that $\exists \pi$ for which $\delta(\pi, q_M) = x$ and $\delta(\pi, \widehat{q_M}) = \widehat{x}$, $\theta(x) \sqsubseteq \theta(\widehat{x})$. In other words, extending the rule extends the information states of its categories monotonically. A similar relation exists between D and $\widehat{D}$. The set of all nodes x in M such that $\exists \pi$ for which $\delta(\pi, q_M) = x$ and $\delta(\pi, \widehat{q_M}) = \widehat{x}$ will be denoted by $[\widehat{x}]^{-1}$ (and likewise for nodes in D). There may be more than one node in $[\widehat{x}]^{-1}$ because of unifications that occur during the extension of M to $\widehat{M}$.

3 The Indexing Timeline

Indexing can be applied at several moments during parsing. We introduce a general strategy for indexed parsing, with respect to what actions should be taken at each stage.

Three main stages can be identified. The first one consists of indexing actions that can be taken off-line (along with other optimizations that can be performed at compile-time). The second and third stages refer to actions performed at run time.

Stage 1. In the off-line phase, a static analysis of grammar rules can be performed. The complete content of mothers and daughters may not be accessible, due to variables that will be instantiated during parsing, but various sources of information, such as the type signature, appropriateness specifications, and the types and features of mother and daughter descriptions, can be analyzed and an appropriate indexing scheme can be specified. This phase of indexing may include determining: **(1a)** which daughters in which rules will certainly not unify with a specific mother, and **(1b)** what information can be extracted from categories during parsing that can constitute indexing keys. It is desirable to perform as much analysis as possible off-line, since the cost of any action taken during run time prolongs the parsing time.

Stage 2. During parsing, after a rule has been completed, all variables in the mother have been extended as far as they can be before insertion into the chart. 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.

Stage 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 have been extended as far as they can be before unification with a chart edge. The information identified in stage (1b) can be extracted and unified as a precursor to the remaining steps involved in category unification. These steps also take place at this stage.

4 TFSG Indexing

To reduce the time spent on failures when searching for an edge in the chart, each edge (edge's category) has an associated index key which uniquely identifies the set of daughter categories that can potentially match it. When completing a rule, edges unifying with a specific daughter are searched for 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. When a rule is completed, its mother M is added to the chart according to M's *indexing scheme*, which is the set of index keys of daughters that might possibly unify with M. The index is implemented as a hash, where the hash function applied to a daughter yields the daughter's index key (a selection of chart edges). For a passive edge representing M, M's indexing scheme provides the collection of hash entries where it will be added.

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

4.1 Positional Indexing

In positional indexing, the index key for each daughter is represented by its position (rule number and daughter position in the rule). The structure of the index can be determined at compile-time (first stage). For each mother M in the grammar, a collection $\mathcal{L}(M) = \{(R_i, D_j) |$ daughters that can match $M\}$ is created (M's indexing scheme), where each element of $\mathcal{L}(M)$ 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 M.

For TFSGs it is not possible to compute off-line the exact list of mother-daughter matching pairs, but it is possible to rule out certain non-unifiable pairs before parsing — a compromise that pays off with a very low index management time.

During parsing, each time an edge (representing a rule's mother M) is added to the chart, it is inserted into the hash entries associated with the positions (R_i, D_j) from the list $\mathcal{L}(M)$ (the number of entries where M is inserted is $|\mathcal{L}(M)|$). The entry associated with the key (R_i, D_j) will contain only categories that can possibly unify with the daughter at position (R_i, D_j) in the grammar.

Because our parsing algorithm closes categories depth-first under leftmost daughter matching, only daughters D_i with $i \geq 2$ are searched for in the chart (and consequently, indexed). We used the EFD-based modification of this algorithm (Penn and Munteanu, 2003), which needs no active edges, and requires a constant two copies per edges, rather than the standard one copy per retrieval found in Prolog parsers. Without this, the cost of copying TFS categories would have overwhelmed the benefit of the index.

4.2 Path Indexing

Path indexing is an extension of positional indexing. Although it shares the same underlying principle as the path indexing used in automated reasoning (Ramakrishnan et al., 2001), its functionality is related to quick check: extract a vector of types from a mother (which will become an edge) and 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 that it identifies these paths by a static analysis of grammar rules, performed off-line and with no training required. Path indexing is also built on top of positional indexing, therefore the vector of types can be different for each potentially unifiable mother-daughter pair.

4.2.1 Static Analysis of Grammar Rules

Similar to the abstract interpretation used in program verification (Cousot and Cousot, 1992), the static analysis tries to predict a run-time phenomenon (specifically, unification failures) at compile-time. It tries to identify nodes in a mother that carry no relevant information with respect to unification with a particular daughter. For a mother M unifiable with a daughter D, these nodes will be grouped in a set $StaticCut(M,D)$. Intuitively, these nodes can be left out or ignored while computing the unification of $\widehat{M}$ and $\widehat{D}$. The $StaticCut$ can be divided into two subsets: $StaticCut(M,D) = RigidCut(M,D) \cup VariableCut(M,D)$.

The $RigidCut$ represents nodes that can be left out because neither they, nor one of their δ_π-ancestors, can have their type values changed by means of external variable sharing. The $VariableCut$ represents nodes that are either externally shared, or have an externally shared ancestor, but still can be left out.

Definition 2. *$RigidCut(M,D)$ is the largest subset of nodes $x \in M$ such that, $\forall y \in D$ for which $x \bowtie y$:*
1. $x \notin Ext(M)$, $y \notin Ext(D)$,
2. $\forall x' \in M$ s.t. $\exists \pi$ s.t. $\delta(\pi, x') = x$, $x' \notin Ext(M)$, and
3. $\forall y' \in D$ s.t. $\exists \pi$ s.t. $\delta(\pi, y') = y$, $y' \notin Ext(D)$.

Definition 3. *$VariableCut$ is the largest subset of nodes $x \in M$ such that:*
1. *$x \notin RigidCut(M,D)$, and*
2. *$\forall y \in D$ for which $x \bowtie y$, $\forall s \sqsupseteq \theta(x), \forall t \sqsupseteq \theta(y)$, $s \sqcup t$ exists.*

In words, 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 D can have. Due to structure sharing, the types of nodes in M and D can change during parsing, by being specialized to one of their subtypes. Condition 2 ensures that the types of these nodes will remain compatible (have a least upper bound), even if they specialize during rule completion. An intuitive example (real-life examples cannot be reproduced here — a category in a typical TFSG can have hundreds of nodes) is presented in Figure 4.

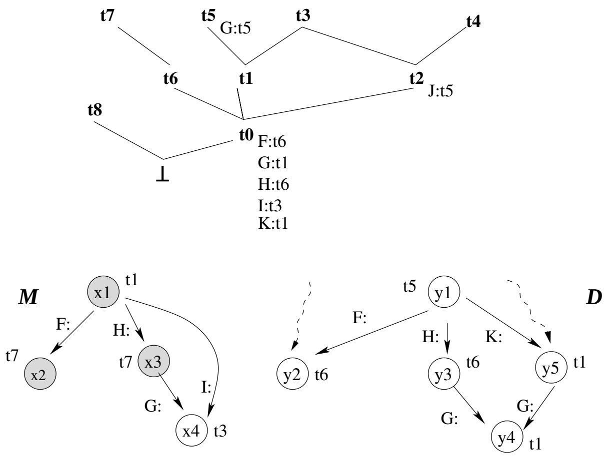

Figure 4: Given the above type signature, mother M and daughter D (externally shared nodes are pointed to by dashed arrows), nodes x_1, x_2, and x_3 from M can be left out when unifying M with D during parsing. x_1 and x_3 $\in RigidCut(M,D)$, while $x_2 \in VariableCut(M,D)$ ($\theta(y_2)$ can promote only to t_7, thus x_2 and y_2 will always be compatible). x_4 is not included in the $StaticCut$, because if $\theta(y_5)$ promotes to t_5, then $\theta(y_4)$ will promote to t_5 (not unifiable with t_3).

When computing the unification between a mother and a daughter during parsing, the same outcome (success or failure) will be reached by using a reduced representation of the mother ($\widehat{M}^{\rtimes_D}$), with nodes in $StaticCut(M,D)$ removed from $\widehat{M}$.

Proposition 1. *For a mother M and a daughter D, if $M \sqcup D\downarrow$ before parsing, and $\widehat{M}$ (as an edge in the chart) and $\widehat{D}$ exist, then during parsing: (1) $\widehat{M}^{\rtimes_D} \sqcup \widehat{D}\downarrow \Rightarrow \widehat{M} \sqcup \widehat{D}\downarrow$, (2) $\widehat{M}^{\rtimes_D} \sqcup \widehat{D}\uparrow \Rightarrow \widehat{M} \sqcup \widehat{D}\uparrow$.*

Proof. The second part ($\widehat{M}^{\rtimes_D} \sqcup \widehat{D}\uparrow \Rightarrow \widehat{M} \sqcup \widehat{D}\uparrow$) of Proposition 1 has a straightforward proof: if $\widehat{M}^{\rtimes_D} \sqcup \widehat{D}\uparrow$, then $\exists \widehat{z} \in \widehat{M}^{\rtimes_D} \cup \widehat{D}$ such that $\neg\exists t$ for which $\forall \widehat{x} \in [\widehat{z}]_{\bowtie}, t \sqsupseteq \theta(\widehat{x})$. Since $\widehat{M}^{\rtimes_D} \subseteq \widehat{M}$, $\exists \widehat{z} \in \widehat{M} \cup \widehat{D}$ such that $\neg\exists t$ for which $\forall \widehat{x} \in [\widehat{z}]_{\bowtie}, t \sqsupseteq \theta(\widehat{x})$, and therefore, $\widehat{M} \sqcup \widehat{D}\uparrow$.

The first part of the proposition will be proven by showing that $\forall \widehat{z} \in \widehat{M} \cup \widehat{D}$, a consistent type can be assigned to $[\widehat{z}]_{\bowtie}$, where $[\widehat{z}]_{\bowtie}$ is the set of nodes in $\widehat{M}$ and $\widehat{D}$ equivalent to $\widehat{z}$ with respect to the unification of $\widehat{M}$ and $\widehat{D}$.[1]

Three lemmata need to be formulated:

Lemma 1. *If $\widehat{x} \in \widehat{M}$ and $x \in [\widehat{x}]^{-1}$, then $\theta(\widehat{x}) \sqsupseteq \theta(x)$. Similarly, for $\widehat{y} \in \widehat{D}$, $y \in [\widehat{y}]^{-1}$, $\theta(\widehat{y}) \sqsupseteq \theta(y)$.*

Lemma 2. *If types $t_0, t_1, \ldots, t_n$ are 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 $\forall i \in (1, \ldots, n)$, $t \sqsupseteq t_i$.*

[1]Because we do not assume inequated TFSs (Carpenter, 1992) here, unification failure must result from type inconsistency.

Lemma 3. *If* $\widehat{x} \in \widehat{M}$ *and* $\widehat{y} \in \widehat{D}$ *for which* $\widehat{x} \bowtie \widehat{y}$*, then* $\exists x \in [\widehat{x}]^{-1}, \exists y \in [\widehat{y}]^{-1}$ *such that* $x \bowtie y$*.*

In proving the first part of Proposition 1, four cases are 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 A is trivial, and D is a generalization of B and C.

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 $t \sqsupseteq \theta(\widehat{x})$.

Subcase B.i: $\widehat{x} \in \widehat{M}, \widehat{x} \notin \widehat{M}^{\rtimes_D}$. $\forall \widehat{y} \in [\widehat{z}]_{\bowtie} \cap \widehat{D}$, $\widehat{y} \bowtie \widehat{x}$. Therefore, according to Lemma 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 3, $\forall s \sqsupseteq \theta(y), \forall t \sqsupseteq \theta(x)$, $s \sqcup t \downarrow$. But according to Lemma 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 Lemma 2, $\exists t \sqsupseteq \theta(\widehat{x}), \forall \widehat{y} \in [\widehat{z}]_{\bowtie} \cap \widehat{D}$, $t \sqsupseteq \theta(\widehat{y})$.

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

Case C. It will be shown that $\exists t \sqsupseteq \theta(\widehat{y})$ such that $\forall \widehat{x} \in [\widehat{z}]_{\bowtie}$, $t \sqsupseteq \theta(\widehat{x})$. Let $\{\widehat{y}\} = [\widehat{z}]_{\bowtie} \cap \widehat{D}$. The set $[\widehat{z}]_{\bowtie} \cap \widehat{M}$ can be divided into two subsets: $S_{ii} = \{\widehat{x} \in [\widehat{z}]_{\bowtie} \cap \widehat{M} | \widehat{x} \in \widehat{M}^{\rtimes_D}\}$, and $S_i = \{\widehat{x} \in [\widehat{z}]_{\bowtie} \cap \widehat{M} | \widehat{x} \in \widehat{M}, \widehat{x} \notin \widehat{M}^{\rtimes_D}$, and $x \in VariableCut(M,D)\}$. If x were in $RigidCut(M,D)$, then necessarily $|[\widehat{z}]_{\bowtie} \cap \widehat{M}|$ would be 1. Since $S_{ii} \subseteq \widehat{M}^{\rtimes_D}$ and $\widehat{M}^{\rtimes_D} \sqcup \widehat{D} \downarrow$, then $\exists t' \sqsupseteq \theta(\widehat{y})$ such that $\forall \widehat{x} \in S_{ii}, t' \sqsupseteq \theta(\widehat{x})$ (*). However, $\forall \widehat{x} \in S_{ii}$, $\widehat{x} \bowtie \widehat{y}$. Therefore, according to Lemma 3, $\forall \widehat{x} \in S_{ii}, \exists x \in [\widehat{x}]^{-1}, \exists y \in [\widehat{y}]^{-1}$ such that $x \bowtie y$. Thus, since $x \in VariableCut(M,D)$, Condition 2 of Definition 3 holds, and therefore, according to Lemma 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, $\forall s_2' \sqsupseteq t', s_2' \sqcup \theta(\widehat{x}) \downarrow$. Thus, according to Lemma 2 and to (*), $\exists t \sqsupseteq t' \sqsupseteq \theta(\widehat{y})$ such that $\forall \widehat{x} \in S_{ii}, t \sqsupseteq \theta(\widehat{x})$. Thus, $\exists t$ such that $\forall \widehat{x} \in [\widehat{z}]_{\bowtie}$, $t \sqsupseteq \theta(\widehat{x})$. □

While Proposition 1 could possibly be used by grammar developers to simplify TFSGs themselves at the source-code level, here we only exploit it for internally identifying index keys for more efficient chart parsing with the existing grammar. There may be better static analyses, and better uses of this static analysis. In particular, future work will focus on using static analysis to determine smaller representations (by cutting nodes in Static Cuts) of the chart edges themselves.

4.2.2 Building the Path Index

The indexing schemes used in path indexing are built on the same principles as those in positional indexing. The main difference is the content of the indexing keys, which now includes a third element. 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* (as in positional indexing), while $V_{i,j}$ is the *path index vector* containing type values extracted from M. A different set of types is extracted for each mother-daughter pair. So, path indexing uses a two-layer indexing method: the positional key for daughters, and types extracted from the typed feature structure. Each daughter's index key is now given by $\mathcal{L}(D_j) = \{(R_i, V_{i,j})\}$, where R_i is the rule number of a potentially matching mother, and $V_{i,j}$ is the path index vector containing types extracted from D_j.

The types extracted for the indexing vectors are those of nodes found at the end of *indexing paths*. A path π is an indexing path for a mother-daughter pair (M,D) iff: (1) π is defined for both M and D, (2) $\exists x \in StaticCut(M,D), \exists f$ s.t. $\delta(f,x) = \delta(\pi, q_M)$ (q_M is M's root), and (3) $\delta(\pi, q_M) \notin StaticCut(M,D)$. Indexing paths are the "frontiers" of the non-statically-cut nodes of M.

A similar key extraction could be performed during Stage 2 of indexing (as outlined in Section 3), using $\widehat{M}$ rather than M. We have found that this online path discovery is generally too expensive to be performed during parsing, however.

As stated in Proposition 1, the nodes in $StaticCut(M,D)$ do not affect the success/failure of $\widehat{M} \sqcup \widehat{D}$. Therefore, the types of first nodes not included in $StaticCut(M,D)$ along each path π that stems from the root of M and D are included in the indexing key, since these nodes might contribute to the success/failure of the unification. It should be mentioned that the vectors $V_{i,j}$ are filled with values extracted from $\widehat{M}$ after M's rule is completed, and from $\widehat{D}$ after all daughters to the left of D are unified with edges in the chart. As an example, assuming that the indexing paths are THROWER:PERSON, THROWN, and THROWN:GENDER, the path index vector for the TFS shown in Figure 2 is $(\textbf{third}, \textbf{index}, \textbf{neuter})$.

4.2.3 Using the Path Index

Inserting and retrieving edges from the chart using path indexing is similar to the general method presented at the beginning of this section. 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 vector 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.

5 Experimental Evaluation

Two TFSGs were used to evaluate the performance of indexing: a pre-release version of the MERGE grammar, and the ALE port of the ERG (in its final form). MERGE is an adaptation of the ERG which uses types more conservatively in favour of relations, macros and complex-antecedent constraints. This pre-release version has 17 rules, 136 lexical items, 1157 types, and 144 introduced features. The ERG port has 45 rules, 1314 lexical entries, 4305 types and 155 features. MERGE was tested on 550 sentences of lengths between 6 and 16 words, extracted from the Wall Street Journal annotated parse trees (where phrases not covered by MERGE's vocabulary were replaced by lexical entries having the same parts of speech), and from MERGE's own test corpus. ERG was tested on 1030 sentences of lengths between 6 and 22 words, extracted from the Brown Corpus and from the Wall Street Journal annotated parse trees.

Rather than use the current version of ALE, TFSs were encoded as Prolog terms as prescribed in (Penn, 1999a), where the number of argument positions is the number of colours needed to colour the feature graph. This was extended to allow for the enforcement of type constraints during TFS unification. Types were encoded as attributed variables in SICStus Prolog (Swedish Institute of Computer Science, 2004).

5.1 Positional and path indexing evaluation

The average and best improvements in parsing times of positional and path indexing over the same EFD-based parser without indexing are presented in Table 1. The parsers were implemented in SICStus 3.10.1 for Solaris 8, running on a Sun Server with 16 GB of memory and 4 UltraSparc v.9 processors at 1281 MHz. For MERGE, parsing times range from 10 milliseconds to 1.3 seconds. For ERG, parsing times vary between 60 milliseconds and 29.2 seconds.

	Positional Index		Path Index	
	average	best	average	best
MERGE	1.3%	50%	1.3%	53.7%
ERG	13.9%	36.5%	12%	41.6%

Table 1: Parsing time improvements of positional and path indexing over the non-indexed EFD parser.

5.2 Comparison with statistical optimizations

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. However, one of the purposes of non-statistical indexing is to eliminate the burden of training while offering comparable improvements in parsing times. A quick-check parser was also built and evaluated and the set-up times for the indexed parsers and the quick-check parser were compared (Table 2). Quick-check was trained on a 300-sentence training corpus, as prescribed in (Malouf et al., 2000). The training corpus included 150 sentences also used in testing. The number of paths in path indexing is different for each mother-daughter pair, ranging from 1 to 43 over the two grammars.

	Positional Index	Path Index	Quick Check
Compiling grammar	6'30"		
Compiling index	2"	1'33"	-
Training	-	-	3h28'14"
Total set-up time:	6'32"	8'3"	3h34'44"

Table 2: The set-up times for non-statistically indexed parsers and statistically optimized parsers for MERGE.

As seen in Table 3, quick-check alone surpasses positional and path indexing for the ERG. However, it is outperformed by them on the MERGE, recording slower times than even the baseline. But the combination of quick-check and path indexing is faster than quick-check alone on both grammars. Path indexing at best provided no decrease in performance over positional indexing alone in these experiments, attesting to the difficulty of maintaining efficient index keys in an implementation.

	Positional Indexing	Path Indexing	Quick Check	Quick + Path
MERGE	1.3%	1.3%	-4.5%	-4.3%
ERG	13.9%	12%	19.8%	22%

Table 3: Comparison of average improvements over non-indexed parsing among all parsers.

The quick-check evaluation presented in (Malouf et al., 2000) uses only sentences with a length of at most 10 words, and the authors do not report the set-up times. Quick-check has an additional advantage in the present comparison, because half of the training sentences were included in the test corpus.

While quick-check improvements on the ERG confirm other reports on this method, it must be

Grammar	Successful unifications	Failed unifications				Failure rate reduction (vs. no index)		
		EFD non-indexed	Positional Index	Path Index	Quick Check	Positional Index	Path Index	Quick Check
MERGE	159	755	699	552	370	7.4%	26.8%	50.9%
ERG	1078	215083	109080	108610	18040	49.2%	49.5%	91.6%

Table 4: The number of successful and failed unifications for the non-indexed, positional indexing, path indexing, and quick-check parsers, over MERGE and ERG (collected on the slowest sentence in the corresponding test sets.)

noted that quick-check appears to be parochially very well-suited to the ERG (indeed quick-check was developed alongside testing on the ERG). Although the recommended first 30 most probable failure-causing paths account for a large part of the failures recorded in training on both grammars (94% for ERG and 97% for MERGE), only 51 paths caused failures at all for MERGE during training, compared to 216 for the ERG. Further training with quick-check for determining a better vector length for MERGE did not improve its performance.

This discrepancy in the number of failure-causing paths could be resulting in an overfitted quick-check vector, or, perhaps the 30 paths chosen for MERGE really are not the best 30 (quick-check uses a greedy approximation). In addition, as shown in Table 4, the improvements made by quick-check on the ERG are explained by the drastic reduction of (chart look-up) unification failures during parsing relative to the other methods. It appears that nothing short of a drastic reduction is necessary to justify the overhead of maintaining the index, which is the largest for quick-check because some of its paths must be traversed at run-time — path indexing only uses paths available at compile-time in the grammar source. Note that path indexing outperforms quick-check on MERGE in spite of its lower failure reduction rate, because of its smaller overhead.

6 Conclusions and Future Work

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. Our experimental evaluation demonstrates that indexing by static analysis is a promising alternative to optimizing parsing with TFSGs, although the time consumed by on-line maintenance of the index is a significant concern — echoes of an observation that has been made in applications of term indexing to databases and programming languages (Graf, 1996). Further work on efficient implementations and data structures is therefore required. Indexing by static analysis of grammar rules combined with statistical methods also can provide a higher aggregate benefit.

The current static analysis of grammar rules used as a basis for indexing does not consider the effect of the universally quantified constraints that typically augment the signature and grammar rules. Future work will investigate this extension as well.

References

B. Carpenter and G. Penn. 1996. Compiling typed attribute-value logic grammars. In H. Bunt and M. Tomita, editors, *Recent Advances in Parsing Technologies*, pages 145–168. Kluwer.

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

P. Cousot and R. Cousot. 1992. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2–3).

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

D. Flickinger. 1999. The English Resource Grammar. http://lingo.stanford.edu/erg.html.

P. Graf. 1996. *Term Indexing*. Springer.

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

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

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

G. Penn. 1999a. An optimised Prolog encoding of typed feature structures. Technical Report 138, SFB 340, Tübingen.

G. Penn. 1999b. Optimising don't-care non-determinism with statistical information. Technical Report 140, SFB 340, Tübingen.

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

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

Swedish Institute of Computer Science. 2004. SICStus Prolog 3.11.0. http://www.sics.se/sicstus.