Array TFS storage for unification grammars

Glenn C. Slayden

A thesis submitted in partial fulfillment of the
requirements for the degree of

Master of Science

University of Washington
2012

Committee:
Emily M. Bender
Stephan Oepen

Program Authorized to Offer Degree:
Computational Linguistics

Abstract

Constraint-based grammar formalisms such as Head-Driven Phrase Structure Grammar (HPSG) model linguistic entities as sets of attribute-value tuples headed by types drawn from a connected multiple-inheritance hierarchy. These *typed feature structures* (TFSes) describe directed graphs, allowing for the application of graph-theoretic analyses. In particular, graph *unification*—the computation of the most general structure that is consistent with a set of argument graphs (if such a structure exists)—can be interpreted as expressing the satisfiability and combination functions for the represented linguistic entities, thus providing a principled method for describing syntactic elaboration. In competent natural language grammars, however, the graphs are typically large and numerous, and computational efficiency is a key engineering concern. This thesis describes a method for the storage of typed feature structures where each TFS comprises a self-contained, contiguous memory allocation with a tabular internal structure. Also detailed is an efficient unification algorithm for this storage mode. The techniques are evaluated in *agree*, a new managed-execution concurrent unification chart parser which supports both syntactic analysis (parsing) and surface realization (generation) within the framework of the DELPH-IN (Deep Linguistic Processing with HPSG Initiative) joint reference formalism.

Acknowledgements

Thanks to my advisor, Emily Bender, and to my reader Stephan Oepen. Remaining errors in the text are the fault of myself alone. Thanks also to Ann Copestake and Dan Flickinger for their inspiration and guidance. Although not described in this thesis, the *agree* generation and transfer components are extensions of this work which could not have been completed without the insights and development skills of Spencer Rarrick. Woodley Packard also provided encouragement and helpful advice regarding the generator. I completed additional work on *agree* while I was at King Mongkut's University of Technology in Thonburi, and I thank my colleagues there, especially Nuttanart Facundes, for the helpful discussions. Mostly, I thank my family and friends for their support and love.

In memory of Laurie Poulson

1 Introduction

Constraint-based linguistic formalisms use hierarchical sets of attribute-value tuples—feature structures—to represent linguistic entities. When each such structure is associated with a type drawn from a carefully arranged multiple-inheritance hierarchy, and when the value of every attribute-value tuple is another such *typed feature structure* (TFS), an elegant system for declaratively modeling natural language arises (Carpenter 1992).

The suitability of this arrangement is in part due to an equivalent interpretation of feature structures as directed graphs, which allows graph-theoretical analysis to be applied to the model. In particular, graph *unification*—the computation of the most general graph that preserves all of the information in a set of conjoined argument graphs (if such a graph exists)—expresses the satisfiability and combination functions for the represented linguistic entities, providing a principled method for modeling the elaboration of linguistic structure.

The unification of directed graphs representing linguistic TFSes is the most costly computational operation in parsing and generating with unification grammars (Callmeier 2001, 39). Processing a simple sentence with a competent natural language grammar can involve hundreds of thousands of unification operations, each operating on graphs of several hundred nodes or more. This intense computational sink has motivated considerable research into algorithms for efficient graph unification.

Unification algorithms require intermediate structure to be built during the course of unification. With realistic grammars, unification failures far outnumber successful unifications during parsing (Tomabechi 1991), and any structure created up to the point of detecting the failures is discarded. Efficiently discarding these unwanted structures has been an important theme in the unification literature. Wroblewski (1987) introduced the standard and effective technique of using a generation counter to categorically invalidate a large swath of nodes. Tomabechi (1991, 1992) improved on this work, describing an algorithm which exhibits no excess copying; accordingly, this method is used by many modern unification parsers.

Central to the discarding problem is the fact that the most naïve implementation of a TFS is as a set of disjoint nodes, with each node independently allocated from the process heap. Although such a *per-node* implementation affords the advantage that the memory address of each node is a globally perfect hash, the lack of an intermediate node-grouping structure complicates the bulk-discarding task. In this thesis, I also observe that node-centric modeling is at variance with Carpenter's seminal formal analysis of feature structures, which accords a central role to only the structure's topmost—or *root*—node.

Engineering considerations affect unifier performance. Although a per-node design can be implemented in either a native- or managed-execution environment, the method has increased penalties in the latter. In some managed environments, automatic garbage collection introduces additional indirection between object references and their underlying memory; these require

extra time to traverse each time the object is referenced. Creating a distinct garbage-collected object for each of the millions of TFS nodes that occur during parsing or generation can result in considerable overhead.

Nevertheless, managed execution environments remain alluring, owing to the ease of development and comprehensive runtime infrastructure they provide. Given the preceding observations, a managed-code implementation of graph unification may obtain improved performance by departing from per-node allocation. This thesis presents a technique for representing the complete set of nodes for a given typed feature structure within a single distinct array. Such an approach trivializes the discarding problem and reduces the number of garbage collected objects, allowing a managed-execution system to enter the performance realm of native code.

To demonstrate that array storage is effective requires that it be challenged with realistic application scenarios. The techniques described here are implemented in *agree*, a new grammar engineering environment and chart analyzer that is introduced as part of this work. The software supports a number of specialized features and optimizations described in the TFS parsing literature, and is compatible with the technical infrastructure and established grammatical practice of the DELPH-IN (Deep Linguistic Processing with HPSG Initiative)[1] collaborative effort. This international consortium is a research community united in pursuing deep natural language processing with linguistically-motivated precision computational grammars based on Head-Driven Phrase Structure Grammar (HPSG, Pollard and Sag 1994). DELPH-IN compatible declarative grammars permit bidirectional mapping between semantic representations—in the format of Minimal Recursion Semantics (MRS, Copestake et al. 2005)—and surface realizations (i.e., sentences).

The thesis is structured as follows. In Chapter 3, I review typed feature structure formalisms, focusing specifically on the DELPH-IN joint reference formalism (Copestake 2002a). The chapter continues with a description of per-node TFS storage—describing how it corresponds most closely with a graph-theoretic view of the TFS node. I then compare this prevailing method with a new method, the main contribution of this thesis. In this method, a synthetic allocation domain—essentially a specialized allocation space private to each TFS—is superimposed upon the array primitive provided by the runtime environment. The detailed presentation includes a formal definition, contrasts with the traditional approach, illustrative examples, and a discussion of motivating considerations.

Chapter 4 describes a unification algorithm adapted to array TFS storage. I begin with a review of the historical progression of research on algorithms for linguistic TFS unification. Next, I present the new unifier, originally inspired by the Tomabechi (1991) method but adapted for array storage, enabled for concurrency, and with an important simplification. Specifically, the unifier obtains a guarantee not available to previous methods, namely, that the result structure

[1] http://www.delph-in.net

can always be described by simple (scalar-valued) forwarding between the pre-existing set of arity maps which describe the input structures. The operation of the array storage unification algorithm is also demonstrated via a sequence of diagrams illustrating the internal states of the unifier as it progresses through a simple TFS unification operation.

In Chapter 5, I evaluate array TFS storage and the array storage unifier, comparing *agree* to the LKB (Copestake 2002b) and PET (Callmeier 2000) parsers, which represent broadly-delineated managed- and native-execution performance baselines, respectively. In these end-to-end tests, the new system joins the performance class corresponding to the faster of those two systems. In particular, *agree's* built-in concurrency and relatively unfettered throughput scaling predispose it to the real-time parsing of long, complex sentences. Under this condition, evaluation shows parse times for the most complex sentence in the test corpus improved by up to 24% versus the comparison system.

Continuing work is discussed at several points throughout this thesis, and specifically in Sections 3.4, 4.4 and 5.2. Conclusions can be found in Chapter 6.

2 Typed feature structures

This chapter discusses typed feature structures (TFS) and their application in constraint-based grammars. The discussion is divided into three parts. The first part reviews prior work in the formal description of TFSes. Special attention is given to elective aspects of the formalism which have been adopted within the DELPH-IN research community. More to the point, aspects of the formalism—such as disjunction and default constraints—which the consortium's research program eschews, are not reviewed.

To place the new storage method in context, Section 2.2 introduces some of the engineering issues relevant to TFS storage. For example, any representation scheme for TFS nodes must arrange the information from each node distinctly with respect to some singleton domain. A natural choice for this domain is the memory address space of the process. In this simple scheme, each node is sufficiently identified by a pointer, and a TFS can be identified by a pointer to one of these nodes, the *root node* of the TFS.

The fundamental problem of TFS storage is to encode mixed-arity data in a compact manner that permits extremely efficient access. An overview of this *feature arity problem*—the storage and efficient manipulation of typed feature structures used in linguistic modeling—is also given in Section 2.2.

The final part of this chapter, Section 2.3, gives a formal description of TFS storage, a specialized storage and retrieval abstraction over system memory. This storage mode, a key contribution of this thesis, is formally described using notation adopted from the relational algebra (Codd 1970).

It is important to state at the start, however, that taking relational algebra as descriptive vehicle is in no way meant to suggest that conventional relational databases might present a suitable storage mode for linguistic typed feature structures; the relational algebra expressions in Section 2.3 are understood to describe memory-resident data structures. Linguistic parsing expresses extreme performance requirements which are generally considered the domain of in-memory processing, and so locally attached DRAM—in fact specifically uniform-memory architecture (UMA)—was the sole storage substrate considered in the context of the method described in this thesis. Relational algebra notation was adopted for its facility in describing tabular data. The notation was freely extended to suit the needs of my presentation, with no concern for the significance of these departures to RDBMS theory or practice, and these departures—for example, the co-opting of pre-made arity maps from the physical storage layout—likely have no implementation parallel in RDBMS systems.

2.1 TFS formalism

The use of typed feature structures to represent linguistic objects in constraint-based formalisms was most influentially summarized by Carpenter (1992), when he established connections between—and theoretical foundations for—earlier work by Pereira and Warren (1980), Aït-

Kaci (1984), Gazdar et al. (1985), Shieber (1984, 1986), and others. Much of the emerging research of this period was closely related to work in logic programming (Aït-Kaci 1986), lattice computation (Davey and Priestley 1990), and the Prolog computer language (Colmerauer 1986).

In the following sections, I contrast two alternative formal views of typed feature structures. Where possible, I present the notation of the more prominent approach, which is that of Carpenter (1992). As regards notation, one exception—also adopted by Copestake (2002b)—is my use of the symbol $\sqcap$, as opposed to $\sqcup$, for unification, the binary operator which gives the *greatest-lower-bound* (GLB) type. As noted by Copestake (*ibid.*, 55), this seems consistent with a visual presentation of (inverted) trees which increase their specificity downwards.[2] Accordingly, in this thesis the symbol $\sqcup$ represents the *least-upper-bound* (LUB) binary operator. These operators can also be used as unary prefix operators on sets of types.

2.1.1 Type hierarchy and feature appropriateness

Fundamental to typed unification grammars is a single organizing type hierarchy $\langle \textsc{Type}, \sqsubseteq \rangle$, a finite meet semilattice, or bounded complete partial order (BCPO) with a single maximal element $\top \in \textsc{Type}$. That the partial ordering on $\langle \textsc{Type}, \sqsubseteq \rangle$ is complete entails that for any set of compatible types $T \subseteq \textsc{Type}$, there will be a unique GLB type $t = \sqcap T$. This condition is important because we would like the type unification between any two types $s \in \textsc{Type}$ and $t \in \textsc{Type}$, written as $s \sqcap t$, to be deterministic. The type hierarchy can be viewed as a directed graph, and in the DELPH-IN joint reference formalism it is taken to be absent of cycles—a *directed acyclic graph*, or DAG.

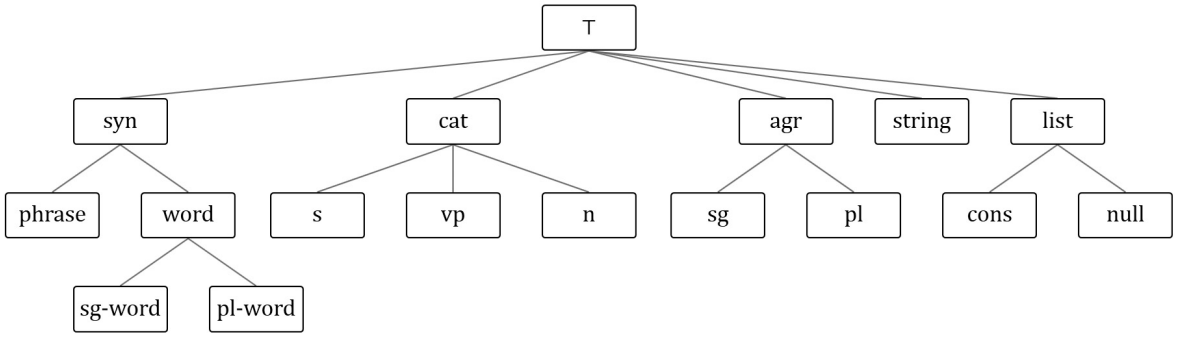


Figure 1. Type hierarchy for a simple grammar[3]

An example of a type hierarchy for a simple grammar is shown in Figure 1. The unique top type $\top \in \textsc{Type}$ subsumes every type in the connected hierarchy. Below this, types are organized so that linguistic generalizations are exposed. In this case, type **syn** subsumes syntactic enti-

[2] The alternate view is justified in Carpenter (1992, 13).

[3] Except where specifically noted, the grammar examples—modified to suit the concerns of this thesis—are adapted from the LKB tutorial grammar 'g5agr' (Copestake 2002b); any errors in these were introduced by my alterations. Diagrams shown in this thesis are produced by *agree*/WPF, one of tools developed for this research.

ties, **cat** subsumes categorical distinctions, and **agr** encodes a singular versus plural distinction. The **list** types support lists of values, and **string** is a special type which subsumes all nodes containing textual values.

Every type in this example hierarchy (except ⊤) has exactly one parent, so this hierarchy does not illustrate multiple inheritance. Multiple inheritance is an expressive technique for grammar writers to exploit, but often the resulting lattice—as authored—will violate the unique lower bound condition between some pair (or pairs) of compatible types.[4] Figure 2 shows a fragment of a type hierarchy taken from the English Resource Grammar (Flickinger 2000) which illustrates the point. The shaded type, which is not included in the grammar as authored, is required to ensure deterministic type unification. As a convenience to grammar writers, implemented systems automatically compute the BCPO of the multiple-inheritance type hierarchy, a process that is briefly described next.

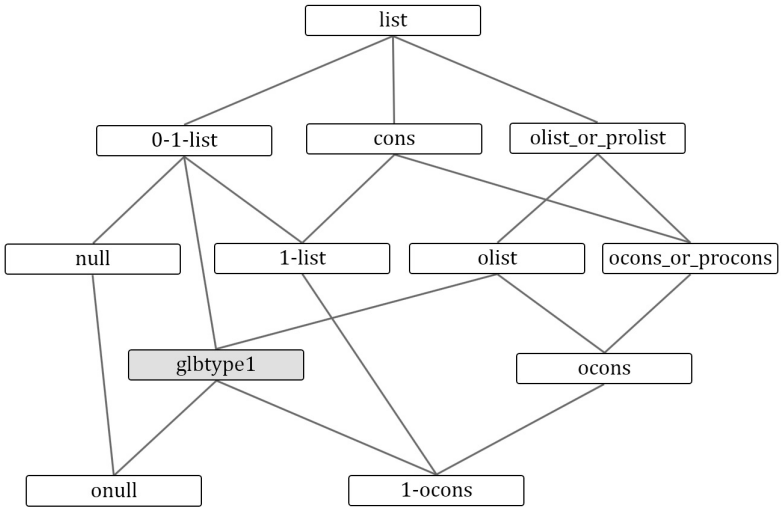


Figure 2 Fragment of the ERG type hierarchy showing a greatest lower bound (GLB) type which was automatically inserted so that the type unification result 0-1-list ⊓ olist is deterministic. Absent glbtype1, the result would be indeterminate, namely, between onull and 1-ocons.

First, a partial order is completed by inserting extra anonymously-named types, as required, to eliminate all cases of non-deterministic type unification between consistent types. The *agree* grammar engineering environment accomplishes this using the technique described in Aït-Kaci (1984). This process may introduce redundant inheritance relationships between types, which should be removed for tidiness and efficiency, so the next step is to compute the transitive reduction of the resulting graph.[5] For this step, rather than reducing the lattice from scratch, *agree* maintains the transitive reduction during and throughout the GLB computation phase.

[4] Specifically, the existence of any pair of distinct types for which there are multiple types which are compatible with both in the pair—yet themselves manifest no subsumption relationship—implies that the partial order of the lattice is incomplete. The undesirable result in the operation of the grammar is that the type unification binary operator is non-deterministic.

[5] The main reason for computing an optimal transitive reduction—the removal of redundant edges—is that a later step in preparing the grammar is to unify each type with the constraints on its parents, a process known as *expanding* the grammar. An optimal transitive reduction ensures that the minimum number of unifications will be performed during expansion. Other than this—and for the convenience of visualizing the type hierarchy with a minimum number of connecting lines—there is no operational use for the reduced graph.

The formalism further extends the type system by introducing zero or more named features $f \in \textsc{Feat}$ for each type in the hierarchy. In the variant presented here (and as adopted by DELPH-IN), each feature must be introduced by exactly one type, $t = \mathit{Intro}(f),\ t \in \textsc{Type}$. By definition, the feature becomes *appropriate* for that type and all the types it subsumes—that is, all of the types which are descendants of t in the type hierarchy. The appropriateness function $\mathit{Approp}(f, t)$ is thus defined as the unique partial function[6] in $\textsc{Feat} \times \textsc{Type}$ that globally satisfies (2.1):

$$\bigwedge_{f \in \textsc{Feat}} \bigwedge_{t \in \textsc{Type}} \bigsqcup f \quad \mathit{Approp}(f,t)\downarrow \tag{2.1}$$

The grammar's set of types and features can be arbitrarily arranged to describe constrained entities. To do so, they are arranged in independent hierarchical structures called *typed feature structures* (TFS), which are formally described in the next section, Section 2.1.2. An example is shown in Figure 3, where the type hierarchy from Figure 1 has been extended by describing a canonical constraint for some of the types. Canonical constraints are authored as TFSes which are headed by the respective type, and which optionally list a set of named properties—the features—which are each paired with a value. Each value in these *feature-value pairs* refers to further substructure, expressed in the form of a nested TFS.[7] The type heading this substructure may have no appropriate features, which signals a leaf in the arbitrarily nested structure. A feature is introduced by exactly one type in $\langle \textsc{Type}, \sqsubseteq \rangle$,

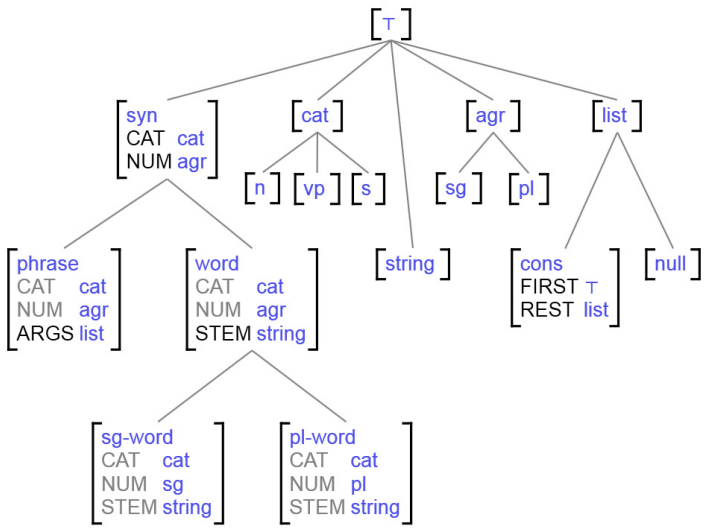


Figure 3 The type hierarchy from Figure 1 is extended by describing canonical constraints for each types. These *typed feature structures* (TFS) optionally pair a set of features with additional constraints. Feature names are shown with black text when they are introduced by their node type, or gray text if they are inherited from a parent type.

[6] Regarding notation for partial functions, the downward arrow suffix $f(g)\downarrow$ defines a total predicate which indicates whether partial function f is defined for argument g. This form for defining the appropriateness function is adopted from Carpenter (1992).

[7] In this thesis, the term "TFS" will generally not be used to refer to the organization of TFS substructure. For the purposes of this presentation, the fact that typed feature structures are structurally recursive is secondary to the engineering considerations related to the lifecycle of top-level structures. Therefore, unless otherwise indicated, TFS refers only to an entire structure whose topmost node is not referenced from within any other TFS (excluding for purposes of a non-linguistic structure sharing optimization). This definition includes all of the structures which comprise a loaded grammar, structures created via unification, parse chart edges (including partial rule applications), etc. To highlight this point, consider rule-daughter unification: the mother and daughter are each TFSes, but the point within the mother at which the daughter is unified will not be called a "TFS," (because it has outer structure relative to this internal node); this use of nomenclature is chosen because it correctly predicts the additional complication these types of unifications present for monolithically stored structures.

an association deduced from context. For each type, the features it introduces plus the features inherited from its parent type (or types) form its set of appropriate features.

Completing the definition of the grammar itself, in constraint-based grammars, the linguistic entities corresponding to the entity classes in the traditional formal grammar tuple $G = \langle N, \Sigma, R, S \rangle$—*viz.*, intermediate syntactic fragments, the lexicon, the rules, and the set of root symbols—are all represented as typed feature structures.

The set of constraints explicitly written for inclusion in the grammar becomes a reference set which, taken as a whole, comprises the intended model of linguistic structure. Constraints can be composed in infinite variation under the operation of TFS unification, which serves as the dual satisfiability and monotonic composition functions in unification grammars. TFS unification produces a new TFS (if such a structure is possible), namely, the necessarily unique most general structure which jointly satisfies all of the constraints in the argument structures; this is the topic of Chapter 4.

Formally, the solution to the grammar's system of simultaneous constraints under unification is the (infinite) set of permissible linguistic structures. In these terms, the ambition of a grammar is to set forth the most succinct set of canonical feature structures which permit, via any sequence of successful unifications, only and exactly those structures which are judged to be grammatical. In practice, exploration of this vast solution space is guided via specialized unification-based chart parsers. The process is computationally intensive; a parser may produce tens of thousands of transient structures during the course of analyzing a sentence.

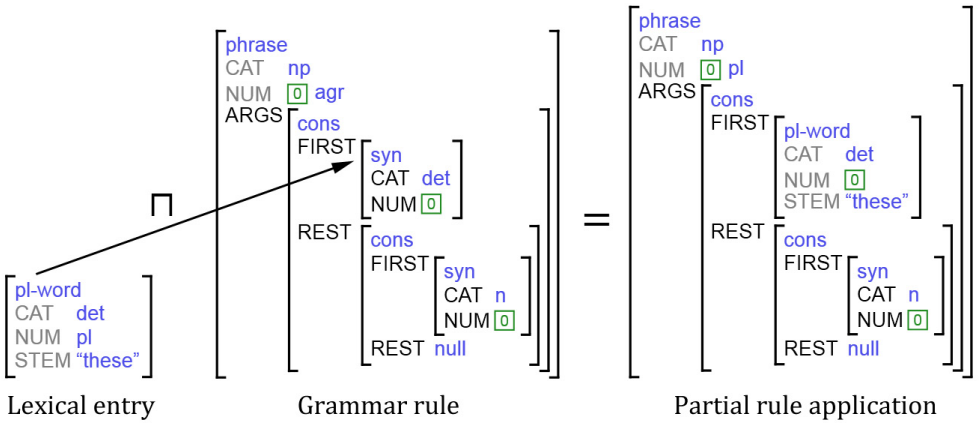


Figure 4. Every linguistic entity in a unification grammar is a TFS. Here, the TFS representing the lexical entry for the word "these" is unified with the first of two surface positions in a rule which forms a noun phrase from a determiner and a noun. The result reflects an incomplete rule application. Note that the type of the NUM feature—which was 'agr' in the rule—has become more constrained ('pl') in the result structure, reflecting that only plural nouns are eligible to complete this parse hypothesis.

An example of how unification composes new structures is shown in Figure 4. In this case, a lexical entry representing the plural determiner "these" is unified with (a subsection of) a rule which forms a noun phrase from a determiner and a noun. The rule TFS belongs to a certain

class of structure that I refer to as *mother-daughter TFS*es; these make available one or more internal substructure nodes ('ARGS' or daughter positions) at which unification operations can be initiated. In this vein, Figure 4 shows a *mother-daughter unification*, which unifies the root node of one TFS with one of the daughter position in a mother-daughter TFS. If successful, the operation yields a result structure whose root coincides with the *mother* TFS.

The typed feature structure defines the manner in which the types and features defined by a grammar are arranged in order to describe arbitrarily constrained linguistic entities. The arrangement is simple: a typed feature structure consists of exactly one type $t \in \textsc{Type}$ and a set of attribute-value pairs which pair each feature in the set

$$\bigcup_{f \in \textsc{Feat},\, f:\mathcal{A}pprop(f,t)\downarrow} f \tag{2.2}$$

—the *attributes*—with another "typed feature structure"—the *value*. Although the elegance of this informally stated definition is compelling, it is not the basis for the most widely cited formal treatment, that of Carpenter (1992). In Sections 2.1.2 and 2.1.3, I compare approaches evolved from (2.2) with the Carpenter's canonical formalism, hereafter the *functional formalism*.

2.1.2 Functional formalism

This section reviews the analysis of typed feature structures advanced by Carpenter (1992). To begin, it is convenient to have a shorthand notation for feature paths within a TFS. These definitions make use of the list-concatenation operator, $\oplus$.

$\textsc{Path} = \textsc{Feat}^*$	the set of all paths	(2.3)
ϵ	the empty path	(2.4)
$\alpha \in \textsc{Path}$	a path[8]	(2.5)
$\alpha = \epsilon \oplus \alpha$	traversal of the empty path is vacuous	(2.6)
$\alpha_i = \langle f_0, f_1, \ldots, f_{n_i} \rangle$	A path is an ordered sequence of features	(2.7)
$f\alpha = \langle f \rangle \oplus \alpha$	path extension	(2.8)

A typed feature structure instance F is defined by Carpenter as follows.

$F = \langle Q, \overline{q}, \theta, \delta \rangle$	typed feature structure F is a 4-tuple of:	(2.9)
$Q: \overline{q} \sqsubseteq q$	a finite set of nodes subsumed by a root node;	(2.10)
$\overline{q} \in Q$	a distinguished root node;	(2.11)
$\theta: Q \rightarrow \textsc{Type}$	node typing function (total);	(2.12)
$\delta: Q \times \textsc{Feat} \rightarrow Q$	feature value function (partial).	(2.13)

[8] To avoid a conflict with the relational algebra projection operator, which is used later in this thesis, paths are designated by α, rather than π, as in Carpenter's presentation

The set of all feature structures is given by $\mathcal{F}$. From a graph perspective, the unappealing aspect of this definition is that the two properties of a node—its type and its feature-value pairs—are bifurcated between θ and δ. The following convenience extensions allow nodes to be identified by their distinct paths from the root node.

$\delta: Q \times \textsc{Path} \rightarrow Q$	the feature value partial function can apply to paths	(2.14)
$\delta(q, \epsilon) = q$	traversing the empty path from any node is vacuous	(2.15)
$\delta(q, f\alpha) = \delta(\delta(f, q), \alpha)$	path traversal can be expressed with recursion	(2.16)

By the end of the 1990s, this formal approach to typed feature structures had become HPSG canon, but an alternative formulation of typed feature structures—as directed graphs—was also well studied. First investigated in separate work by Johnson (1987), Smolka (1989) and King (1989), the graph idea was later expanded by King, who lobbied for its official adoption within the then-emergent HPSG school (King 1994).

As for Carpenter's views, he methodically dismisses the graph approach in his seminal book (Carpenter 1992, discussed further in Section 2.1.4). In time, it was his functional approach that was endorsed by Pollard and Sag (*ibid.*, 1). Also influential was the adoption of this approach by Copestake (2002b) in her book which documents the LKB grammar engineering tool.

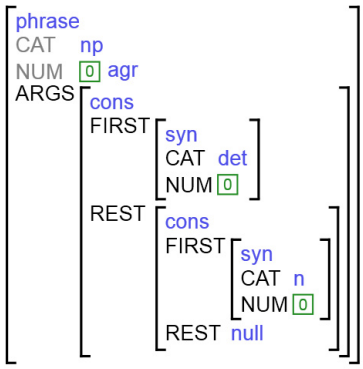
Notwithstanding this, the graph interpretation of the TFS is relevant to the work presented in this thesis, particularly as a fairly literal blueprint of the predominant engineering practice for DAG implementations. The next section gives a graphical formulation of the typed feature structure. Because the fundamental conception is straightforward, my presentation does not recall the detailed formal analyses of Smolka and his contemporaries. Instead, I introduce a new formulation which is intended to focus attention on issues relevant to the main thesis topics which follow.

2.1.3 DAG interpretation

As noted at the beginning of Section 2.1.1, the type hierarchy $\langle \textsc{Type}, \sqsubseteq \rangle$ is equivalently a directed acyclic graph (DAG), where each type is a node, and the transitive closure of the graph exactly expresses the grammar's type unification partial function. Unrelated to this particular graph equivalence, DAGs have further utility in the modeling of typed feature structures. Specifically, the extension of $\langle \textsc{Type}, \sqsubseteq \rangle$ with features $f \in \textsc{Feat}$ means that each TFS also has an equivalent DAG representation. In this correspondence, the graph edges are labeled with features and they represent not an inheritance relationship expressing linguistic generalization, but rather the topological structure of the typed feature structure. Figure 5 illustrates this interpretation.

The figure also illustrates how the backbone of a traditional context-free grammar (CFG) is encoded in TFS grammars. A TFS can contain list-valued values by employing the 'cons cell'

mechanism (adopted from the *LISP* programming language), and the terminals or pre-terminals comprising the right side of a CFG rule can then be encoded as a list of sub-structural TFS positions that correspond to proximal positions in the outermost input structure.[9]



$$NP \rightarrow DET\ N$$

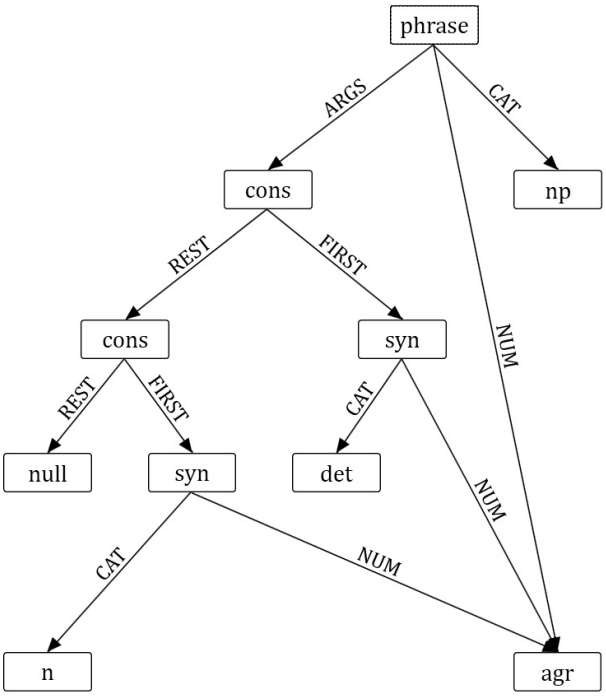


Figure 5. Typed feature structures can be represented as directed acyclic graphs where the types are the nodes and the features are connecting "edges." At right is the DAG representation of the grammar rule depicted (with a conventional linguistic diagram) above. Note that the reentrancy depicted with green boxes in the latter manifests in the graph as multiple DAG edges pointing to the same node.

I refer to any scheme in which TFS substructures ("nodes") are conceived as TFSes—the view carefully avoided by Carpenter—as a *graphical formalism*. Engineering implementations which fundamentally parallel the graph treatment are called *per-node* implementations. Further remarks on the contrast with the functional formalism are presented in Section 2.1.4. In contrast, this thesis proposes a method that fundamentally diverges from per-node modeling. Although the array storage scheme proposed in this thesis does not instead embrace the functional formalism (certain of its aspects remain impractical for direct implementation), in at least one regard, the new work does hew more closely to Carpenter's traditional formulation. Specifically, it incorporates a requirement that the root node of a TFS—*viz*., the topmost linguistically relevant structure—be inherently distinguished from its substructure.

Conceptually, the graph view of the TFS is simple to express. Extending (2.2), in the graph treatment, a node $q \in Q$ is defined as a 2-tuple of a type and a feature-node pairing for each distinct feature appropriate to the type:

$$q_i = \langle t_i \in \textsc{Type}, \bigcup_{f \in \textsc{Feat}, \mathcal{A}\mathit{pprop}(f,t_i)\downarrow} \langle f, q_j \in Q \rangle \rangle \tag{2.17}$$

[9] For parsing, the proximity condition is adjacency (of lexemes) in the surface string ("sentence"). In the case of generation, the list in question is the set of elementary predications (EPs) in the semantics built thus far, and the proximity condition is mutual-exclusion of EPs from the input semantics with this (unordered) set. Other than operating on a proximity condition which is abstracted in this way, the *agree* chart analyzer has little parse *vs*. generate distinction.

In this formulation, q_j is taken to be a *reference* to another node in Q, such that more than one node in Q may include it amongst its feature values. The fact that nodes are modeled in the notational domain in this description makes it tedious to describe specific structures with this type of notation. This issue, a reduced degree of axiomatization (as compared to Carpenter's approach) will be addressed further in the next section.

Note that this recursive definition does not distinguish the modeling of a top-level TFS—used in linguistic modeling—from the modeling of its substructure. Although this characteristic—that nodes and substructure are equally described by (2.17)–does not provide an obvious or inherent way to differentiate top-level linguistic entities from their contents, the simplicity of the conception is nevertheless appealing, as (2.17) alone can be taken as a generally sufficient description of the essence of typed feature structures.[10]

This same characteristic—a purely uniform treatment of sub-structural and root TFS nodes—also facilitates an effective engineering optimization which existing DELPH-IN parsers exploit, and which is not available, in an immediately obvious way, to the new storage method described in this thesis. At issue is the sharing of identical substructure between disjoint TFSes in order to reduce the storage cost for large numbers of TFSes. The technique, known as *structure sharing*, is examined in Section 2.2.2. Observations regarding structure sharing with respect to array storage can be found at the end of that section, and also on page 52, in the discussion of the array storage example.

2.1.4 Contrasting the functional and graph approaches

I now briefly contrast the two dissimilar conceptions of the TFS summarized in the preceding sections. Given the elegance and simplicity of the graph notion—that is, thinking of each TFS node as having a list of tuples of the form $\langle f \in \textsc{Feat}, TFS \rangle$—this section considers why the formalism that appears more cumbersome was adopted, and why Carpenter favored it.

Endorsement by Pollard and Sag signaled a degree of acceptance for Carpenter's formalism that the graph formalism was not able to attract in the linguistic community. One reason that this is surprising is the fact that, as an *engineering* abstraction, the functional formalism is not nearly as compelling—a point mentioned in Sections 2.1.2 and 2.3—and that *de facto* practice in computer engineering encourages a view closer to (2.17), the node-centric graphical conception. Best practices in object-oriented programming gravitate towards simple models in which like behaviors and properties are identified and conflated. Indeed, the internal representation of typed feature structures found in existing unification parsers, such as PET and the LKB, most closely corresponds to the graph abstraction described in this section.

To a certain degree, the elegance of the recursive TFS definition hides co-identity relationships between coreferenced nodes by promoting them to the domain of notation. For example, in

[10] A valid criticism, however, is that while the formulation succinctly and elegantly describes all possible TFSes, it is awkward for describing some particular TFS. This point is addressed in more detail in Section 2.1.4.

(2.17), nodes q_j and q_k are coreferenced if and only if $j = k$. Although fundamentally adequate as a description, (2.17) resists further analysis. Indeed, 'unrolling' the recursive definition by manually writing down the graphical description of some specific TFS would be a tedious process with no reusable result. Carpenter's axiomatized approach to TFS analysis may have garnered wider acceptance because it avoids this problem. Functionally abstracting over TFS topology relegates unruly structural variation to the domain of homogenous modeled data—as opposed to instances of lengthy, opaque singleton formulae—more readily enabling further analysis of the structures' theoretical properties.

It is not necessary to speculate about Carpenter's demurral of the recursive view. As mentioned in Section 2.1.2, Carpenter knew of the graph treatment, and carefully promoted his method by pointing out a subtle case where the two approaches yield inconsistent analyses, his functional analysis giving what he characterized as a more robust interpretation. The divergent case (Carpenter 1992, 39) involves feature structures containing cycles. Although structures containing cycles are not permitted in the DELPH-IN formalism today, other variants which permit self-referential structures were receiving healthy research attention in the early 1990s. One use for cyclical structures cited by Carpenter is in a feature structure which models the liar's paradox sentence.

Specifically, Carpenter observes that co-identifying entire sub-structures, as opposed to co-identifying only individual nodes, as his functional formalism does, complicates the deterministic processing of structures containing cycles. This is because coreferencing in a graph necessarily creates the appearance of identical substructure at each reentrancy. Carpenter interprets this result as meaning that cyclical structures introduce an infinitude of nodes. Conversely, in his functional formalism, the feature value function δ entails that each node in the feature structure corresponds to exactly one unique substructure, even in the face of cyclic structures, and despite the structure having an infinite number of *paths*.

The preceding sections reviewed two approaches to the formal description of TFSes. To complete the discussion of established theoretical practices related to typed feature structures, the next section discusses the TFS well-formedness requirement that many comprehensive formalisms, such as the DELPH-IN joint reference formalism, superimpose upon the fundamental TFS formal entity.

2.1.5 Well-formedness

As discussed in the section 2.1.1, the operational content of a unification grammar is a set of typed feature structures, each belonging to a predefined class of grammar machinery. Relevant classes include grammar rules, start symbols (root conditions), morphological or other lexical rules, lexical entries, and so on.

In the DELPH-IN joint reference formalism, typed feature structures are always well-formed, which means that no part of any TFS may be incompatible with any canonical (authored) con-

straint on the types that it references. As part of the process of loading of a grammar, all of its feature structures are made well-formed. In this process—which is called *expanding* the grammar—the TFS representing the authored constraint for each type is unified with the constraint on its parent types (that is, with each parent's own expanded TFS). Additionally, the internal nodes of each TFS are unified with the expanded TFS for their type.[11]

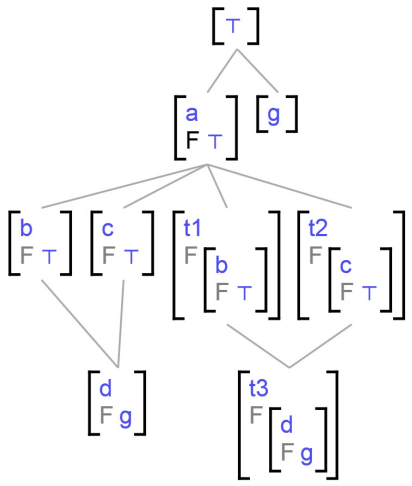


Figure 6. Because unifying compatible types ***b*** and ***c*** results in a distinct type (***d***), the unification of ***t1*** and ***t2*** will introduce an additional unification with the constraint on ***d***, thus introducing ***g*** to ***t3***. Note also from this example that a path can use the same feature—*viz.* ***F.F***—more than once without violating (2.1), nor necessarily introducing a cycle.

An example illustrating the well-formedness requirement is shown in Figure 6. Although neither $t1$ nor $t2$ constrain the type at path $F.F$ in their nodes of type b and c, respectively, their unification results in a structure, $t3$, containing a node of type d. Because elsewhere d is constrained such that feature F must be consistent with g, the well-formedness requirement introduces g into $t3$.

Because the unifier emits new TFSes during the course of grammar operation, it becomes responsible for maintaining the well-formedness condition. Accordingly, well-formedness will be revisited in Section 4.1, where the issue is discussed in the context of unification.

The remainder of this chapter contrasts two engineering approaches to TFS representation in implemented systems: the standard technique and the new method investigated in this thesis. Of the descriptive forms presented so far—the functional and the graphical—both engineering implementations parallel more closely the graphical treatment, but in at least one respect—its incorporation of inherent distinction of the root node of the TFS—the new method evokes aspects of Carpenter's more widely accepted formal treatment.

2.2 Design considerations for TFS storage

This section reviews important design considerations in TFS storage. Most such issues have been studied in the context of TFS unification algorithms. A chronological review of the published literature on the subject can be found in Section 4.2, so the objective of this section is to describe the significant themes which motivate the design of array TFS storage, the description of which immediately follows this material.

[11] For this, I point out an optimization which is not noted in the DELPH-IN literature. The latter describes "each node [being unified] with the [expanded] constraint of the type on each node" (Copestake 1993, 2002b). However, as a TFS is expanded, many of its nodes become collaterally well-formed as a result of unifications with other well-formed TFSes. The *agree* unifier treats the lack of well-formedness as a sort of contagion, introducing the condition only on the root node of each un-expanded definition. As unification proceeds, nodes are trivially 'inoculated' or 'infected,' as appropriate. Combined with the technique of using a single unifier call to simultaneously perform all of the required parent type unifications, this reduces the number of well-formedness unifications needed for grammar expansion.

2.2.1 Node allocation and discarding

Per-node TFS representations are characterized by the direct, individual accessibility of all TFS nodes with respect to some singleton addressing domain. Such a design can be used within the native execution environment of a C/C++ program—where node pointers correspond directly to virtual memory locations—or in the managed environment of a LISP, Java, or C# program—which typically insert a level of indirection between object references and their virtual memory locations. The PET parser (Callmeier 2000) is an example of a native-code parser, where nodes are accessed directly with native pointers, and the LKB system (Copestake 2002b) is an example of a managed a managed-code parser, where nodes are accessed indirectly via handles to garbage-collected memory.

In general, because the feature-value pairs represent mixed-arity data, efficient storage is a challenge. This section introduces some of the concerns, but full examination of this topic is the subject of Section 2.2.3. The node object in a per-node design can refer to its list of feature-value pairs by containment, by reference, or by polymorphism. In a containment design, the list of attribute-value pairs is entirely stored directly within the node, so each node is a relatively self-contained, variably-sized storage entity. In native programs, arbitrarily-sized memory blocks can be allocated for any purpose, and the layout imposed on this memory is entirely defined by the application.

Alternatively, each node is allocated with a fixed size, and thus refers to its list of attribute-value pairs, which are stored separately elsewhere, by pointer or reference. Managed environments provide assorted lists, dictionaries, or maps for this purpose; they can be included by reference or by inheritance. In the former case, a reference to the list or map is stored in the user-defined object, rendering the size of the latter again fixed. For the latter case, list object instances are extended into user-defined objects by selecting the list object type as a base type for the user-defined object type. A benefit of the latter technique is that, depending on the details of the runtime environment, instantiating the derived object may result in just one, rather than two, heap allocations.

Section 2.1.4 noted that the graph interpretation of a TFS corresponds more closely to a per-node TFS representation, because just as node coreferencing is implicit in the memory location of a node in per-node implementations, node coreferencing is implicit in the notational variables of the graph interpretation. Furthermore, engineered systems generally exhibit no correlates to the functions which comprise an axiomatized form. This is all to say that directed graphs seem to naturally model the most straightforward computer-internal representation for TFSes. Such a design adheres to the object-oriented programming tenets of encapsulation and proper abstraction: defining entities that reify the most deterministic local relationships. Since types exhibit a one-to-many relationship with their features, they are reified over features: a node is an entity which associates that type with a set of feature-value tuples by containment, proximity, or distinct reference. Reifying features, on the other hand, is a weaker design be-

cause—feature appropriateness being inherited via the type hierarchy—features are lesser predictors of the range of types with which they may be associated.

The overhead of per-node allocation is minimal in native environments. In a naïve approach, each node is independently allocated from the global process address space. Allocations, once granted, are permanently fixed until discarded, so administrative block headers, if present, are simple—and in the simplest case, no such headers are needed. However, few sophisticated systems use a general-purpose allocator for TFS node storage. One problem is that freeing these allocations can result in undesirable fragmentation of the space available for future allocations and this slows subsequent allocations because longer lists of free blocks must be traversed before a block of adequate size is found. Secondly, when a TFS's constituent nodes are dispersed across memory, each must be individually located and freed. Structurally traversing a TFSes just to discover—and then discard—its nodes is untenable. Recent experiments with *agree* show this to be less efficient than a sequential scan of up to 40% more memory, a result observed in unifier node-counting experiments which will be discussed in Section 4.4.10.

To avoid these problems, sophisticated modern parsers (such as PET) pool the node allocations related to a single TFS into privately-managed high-performance heaps which can be discarded as a whole. This basic characteristic—monolithic encapsulation to facilitate bulk discarding—is incorporated in the array TFS storage design as well, although the details are quite different (and these details ultimately disqualify array TFS storage from being categorized as "per-node" storage). In short, issues that threaten to hamper naïve graph abstractions are straightforwardly mitigated and the per-node abstraction remains aligned with sound, well-understood engineering practice.

In managed execution environments, individual allocations entail a different assortment of penalties. Each allocation may include one or more small administrative blocks which are not co-located with the block itself. The actual value of the reference returned to the application may point to such a control item, and this indirection allows the environment to relocate the actual allocation, if necessary. Relocations occur during garbage collection cycles, which allow free spaces to be re-consolidated, eliminating fragmentation. Unamortized for garbage collection, allocation in a managed environment is faster than with the default (general-purpose) process heap in a typical native program; there is only a single pointer to the beginning of free space, which is simply incremented by the required allocation size.[12] There is no free list to traverse (modulo obscure internal details of the runtime environment). On the other hand, when garbage collection is triggered, a lengthy delay can occur.

A potential disadvantage of per-node storage is that nodes from distinct TFSes may be randomly interspersed across the process heap, an effect which is more pronounced as the heap

[12] In fact, this type of speedy allocation is how sophisticated native-code parsers such as PET implement their private domain-aware heaps.

becomes fragmented. Cache locality may be hindered when a processor has to range across large swaths of memory in order to retrieve co-relevant data. Managed heaps are less prone to fragmentation, but are subject to a different penalty. In managed per-node designs, there is no way to inform the garbage collection system that the lifetimes of the objects which comprise a TFS are bound together, so the garbage collector expends considerable effort ascertaining the set of objects that are eligible for disposal.

2.2.2 Structure sharing

An advantage of any per-node system is that the address of any node is a perfect hash of the node's identity. Because TFS membership is not a property of a node's memory address, sharing substructures between disjoint TFSes is trivially implemented by simply sharing a reference to the same node. The discussion here focuses not on linguistically-motivated reentrancy, but rather on the sharing, for efficiency reasons, of any physical structure that happens to be identical.

Ignoring the issue of finding permissible structure-sharing candidates, when the substructure nodes of every TFS are openly accessible in non-movable process memory, it is trivial for any portion of one TFS to co-opt the substructure of another by simply pointing into it. Structure-sharing schemes appropriate to TFS parsing have been well studied (Malouf et al. 2000, van Lohuizen 2001).

However, this same property—the indifference, or *blitheness*, of a node to the top-level structure or structures to which it belongs—creates problems as well. Most significantly, distinguishing nodes only on the basis of their physical—as opposed to logical—identity places a severe limit on the sharing of substructures between top-level graphs. In a problem known as *spurious structure sharing* (or, in theorem proving, the renaming problem, [Pereira 1985]), co-referenced nodes become incorrectly conflated if the same sub-structure appears more than once amongst all the participants in a unification operation (Malouf et al. 2000). For many unification algorithms, this problem means that any structures that are part of the grammar, or which contain internal coreferencing,[13] must be banned from structure sharing. As an obvious—but expensive—solution to this problem, any time it is necessary to introduce distinct instances of a structure at multiple points within the same TFS, the structure can be copied beforehand.

The structure sharing limitations associated with TFS-blithe node storage can be mitigated by explicitly tracking each node's TFS memberships—within the TFS, within the node, or both—and ensuring that the unifier always distinguishes nodes not just on their memory addresses, but on a joint $\langle TFS, address \rangle$ tuple (van Lohuizen 2001). This enhancement comes naturally to concurrent unifier implementations, which are prohibited from using methods that rely on

[13] A structure G that contain reentrancies $\delta(\bar{q}, \alpha_i) = \delta(\bar{q}, \alpha_j)$ is excluded because if such a structure is introduced into result graph R at node r_i and then again at node r_j, nodes $\delta(r_i, \alpha_i)$ and $\delta(r_j, \alpha_i)$ may become incorrectly conflated.

per-node *in situ* scratch fields, since these would be contended resources. Schemes which assign a detached set of slots to the argument TFSes are, as a bonus, likely to resolve the node-identity problem mentioned here. In fact, van Lohuizen notes exactly this, and the restrictions on structure sharing described in Malouf et al. (2000) are lifted in his concurrent parser. For the same reason, *agree* inherently permits unrestricted sharing of all structures in its unifier, regardless of whether they are grammar-canonical, contain coreferencing, or are multiply-introduced during a given unification operation (a moot point since *agree* currently lacks a sophisticated structure sharing implementation).

Any method for determining a node's-TFS membership, such as the one just described, may facilitate the wholesale discarding of top-level structures as well. This issue was discussed in the previous section. But in general, the ease of structure sharing is inversely related to the sophistication of the scheme used to discard unwanted structures. Short of per-node reference counting, ad-hoc sharing creates dependencies on the co-opted structures which should inhibit their disposal. On the other hand, bulk disposal of a swath of related allocations is facilitated by creating specialized heaps which obtain efficiency by coarsely disregarding this type of detail. This discarding problem is a pervasive theme in high-performance unification parsing and it will be discussed again in Chapter 4. For this section on design considerations for TFS storage, the important note is that, although they trivialize the discarding problem, storage schemes which strongly and independently encapsulate each TFSes' substructure—or worse, make those structures inaccessible—complicate their ability to share arbitrarily between TFSes. Both of these conditions apply to the method contributed by this thesis, and therefore arbitrary structure sharing in a regime of TFSes represented as fully encapsulated, system-managed (movable) arrays is an area for future research.

Having said this, *agree* does currently implement a simple form of structure sharing. When loading the grammar, the system automatically identifies top-level grammar TFSes which can be treated as functionally identical by the unifier, perhaps via specialized feature restriction—or by substitution of just the distinguished type value on their root node. These manipulations are simple for *agree* to detect and deploy. As an example, in the English Resource Grammar[14] (Flickinger 2000), the feature 'RNAME' is used for diagnostic purposes but does not contribute to linguistic analysis. Restricting this feature renders a number of grammar entries functionally identical, modulo their root type. Similar techniques allow *agree* to share (at least once) 653 of the 8450 array storage relations (which each represent an entire TFS body, but not its root *out*-tuple) for the ERG's type hierarchy.

2.2.3 The feature arity problem

The fundamental problem of TFS storage is to efficiently encode mixed-arity data. A TFS typically consists of several hundred nodes, and the number of attribute-value pairs stored in each

[14] Unless otherwise noted, this research is based on SVN trunk revision 10342 (November 29, 2011) of the English Resource Grammar.

of these nodes—and the specific set of attribute identities they exploit—are variable. For complex grammars, the number of pairs associated with each node ranges from zero to over a dozen. Attributes in these pairs are drawn from a set of a couple hundred, but they appear in nodes in a small number of fixed combinations. For example, the English Resource Grammar uses 206 features which appear in only 120 distinct configurations, ranging in size from zero to fourteen features.

In many TFS formalisms, including the DELPH-IN joint reference formalism, each feature must be introduced by exactly one type in the type hierarchy. The feature becomes appropriate for this type and all of its descendants, a condition which is an invariant property of the grammar. In a naïve per-node implementation, details of this invariant aspect of the grammar are implicitly proliferated in the list of feature-value tuples stored with every node, since each value in the list must indicate the feature it constrains. Contained in each node as such, the dispersal of this redundant information increases with the total number of nodes in the TFS. In other words, the mere appearance of a feature f amongst the node's feature-value tuples asserts an appropriateness condition that could have been deduced from the type hierarchy alone.

Callmeier (2000) examines this issue and evaluates alternative methods of feature encoding within the per-node framework. In particular, he notes that, when unifying feature structures, any type unification that results in a type with additional appropriate features may require features in the input nodes to be rearranged, to ensure a position for each appropriate feature of the type of the output node, a process he calls *coercion*. This penalty can be avoided by pre-assigning a unique slot for each feature in every node. As conflicting desiderata, coercion—a time factor—competes with node size—a space factor—in the per-node design. Callmeier studies alternative systems for pre-computing feature layouts for each node type. These fixed-arity schemes try to minimize the number of coercions by guaranteeing that each feature always has the same slot in any type. The obvious and naïve implementation wastefully allocates slots in every TFS, some of which will never be used. An improved scheme pre-computes a partitioning of types into sets that do not interfere with each other, and re-uses a smaller number of feature slots on this basis. A third plan, the most space-efficient, gives each type its own feature layout, while attempting to align feature configurations where possible. A similar scheme (adapted for the array storage method described in this thesis) is described in Section 3.4. Callmeier's evaluation shows that, although the third approach is the best-performing fixed-arity scheme, it does not significantly outperform the naïve method, where each feature-value tuple explicitly carries a (numerical) feature identifier.

The fixed-arity schemes evaluated by Callmeier are attempts to address a fundamental problem with the intuitive type-centric TFS representation, namely, that it fails to fully capitalize on the invariant configuration of the grammar's directed type hierarchy. As noted by Carroll (1993, 40), systems can be designed to take advantage of the fact that a fixed subset of features are permitted on a node.

Feature coercion during type unification arises in systems where a typed node contains attribute-value pairs which bind features to nodes. This is the fundamental conception of per-node storage: consider a TFS path as a hierarchical sequence of node-feature alternations $(q, f, q, f, \ldots)$,[15] where each feature f is bound *rightwards* to a node q:

$$(q, (f\ q), (f\ q), \ldots) \tag{2.18}$$

In an alternative approach, the feature is bound *leftwards* in the sequence, giving a model where each node-feature key addresses a subsequent node:

$$\big((q\ f), (q\ f), (q\ f) \ldots\big) \tag{2.19}$$

In the abstract, this subtle reformulation underlies the approach of array TFS storage, the main contribution of this thesis. Essentially, the method pushes the feature arity problem into the domain of node addressing. Integrating domain-specific knowledge—namely, details of the grammar's feature appropriateness condition—into the node addressing scheme means that the maximally efficient native addressing (pointers) can no longer be used—since each address must now encode a contextually determined parameter—but in managed environments direct access is already compromised by additional indirection anyway.

It is not entirely accidental that I depict one additional binding in (2.19) as compared to (2.18). Although the groupings are presented only as an informal conceptual aid, the additional tuple hints at an important result which follows from the new technique, a simplification that can be applied to a well-known TFS unification algorithm. This material is presented later in this thesis, and that presentation will recall this point. Details are given in Section 4.4.1.

It turns out that strongly embedding the feature appropriateness condition into the storage addressing system ultimately results in a simplification of the unification algorithm. This is the subject of Chapter 4, but a summary of the insight is that, if the unifier can be guaranteed that the result TFS is entirely latent within a small, fixed set of fragmentary, mixed-arity feature mappings which are invariant precursors to the operation,[16] then the number of data structures it must maintain and manipulate is reduced. This guarantee is indeed provided—by the mappings which comprise the operation's input arguments themselves. The mixed-arity feature mappings which describe their internal storage layouts are efficient and sufficient for this purpose because the result structure can always be represented by the arrangement of these pre-existing mappings. When these maps are co-opted, the unifier is freed from the requirement that it be able to assemble arbitrary structural fragments in unforeseen ways, resulting in sim-

[15] Taking the features alone from this sequence describes a traditional TFS feature path, as described functionally in (2.7), but for storage purposes the type at each step of the way must also be accommodated, hence this interleaving.

[16] To be precise, the well-formedness requirement entails that some TFSes are not incorporated into the unification operation until it is already underway, but even for these the spirit of the statement holds, because their mappings were invariant prior to—and remain so throughout—the operation.

plification over previous methods. In the end, the unifier work product is reduced to a simple list of scalar values which indicate its sequence of map selections.

At the beginning of this section, I noted that, with per-node storage, the feature arity problem typically entails that redundant traces of the grammar's feature-appropriateness condition accumulate in each node. To conclude this section, I extend the analysis to the conceptually shifted storage mode just introduced. Does it exhibit similar abstraction leakage? In fact, the proposed method still reveals the set of features which are appropriate for the type associated with a node, but the leak is more benign.

The key difference is that in the new conception, the set of features cannot be independently retrieved from a node. To incorporate the feature identity into each node's address, a one-way function—specifically, a hash of the combined node and feature identity—is used, and this prohibits the direct recovery of a node's feature set. To accomplish this, a dependence on the grammar's globally-invariant feature appropriateness condition is fundamentally incorporated in the new access model; see Section 2.3.6, below. By forcing recourse to this single repository, the new model more strongly forfends against features being stored with a type for which they are inappropriate.

To summarize this point, in some designs, per-node storage can exhibit a normalization leak, namely, that two sources of information can exist in conflict with little guidance about which is correct: the presence of a feature-value tuple in a TFS node might be taken as conclusive evidence for the appropriateness of that feature for the type associated with the node—and this could be an error with regard to the global appropriateness condition. With array TFS storage, rogue features can still be stored, but since *access* requires that each caller obtain a feature key from the global invariant itself, later reference to such a feature would require two separate errors.[17]

This concludes the discussion of motivating considerations in the design of a TFS storage system. I now turn to the main contribution of this thesis. The next section presents a formal description of array TFS storage. An implementation of the method is described in Chapter 3.

2.3 Array storage TFS formalism

Carpenter's formal presentation—with its separate, monolithic node-typing and feature-value functions which are only associated with the TFS itself—fares poorly as an engineering abstraction. Because both native and managed execution environments offer a cheap and extremely efficient perfect hash for a node—its memory address or object reference value—it would be quite arcane to consider a literal implementation of the node-typing and feature-value

[17] The normalization issue is raised only as a hypothetical vehicle for illustrating a difference between the two methods. Naturally, implemented systems are designed and tested so that the conditions of the linguistic formalism are correctly enforced at all times and normalization errors cannot be introduced.

functions θ and δ, respectively. Instead, it is obvious programming practice to interconnect a hierarchy of node objects, yielding the per-node implementation of the graph formalism.

But, as noted in Section 2.2.2, adopting the per-node view has problems of its own. Revisiting Carpenter's theoretical model, which more strongly reifies the top-level TFS, may yield insights appropriate to managed implementations. At the outset, discarding from the functional formalism the notion of bifurcated θ and δ functions immediately cuts the lookup overhead in half. Next, we consider the failed-unification discarding penalty; associating each allocation more strongly with the TFS to which it belongs has the potential to trivialize the discarding problem. Details on this are given in Section 4.4.3.

A final benefit of reasserting the primacy of the top-level TFS is the potential for increased structure sharing. Generally, as discussed in Section 2.2.2, encapsulating nodes within larger-scale movable structures inhibits the trivial sharing of substructure, but ironically, the identification of each node with a particular TFS may also facilitate, in the unifier, dealing with shared structures. This is because operating on a $\langle TFS, node \rangle$ association allows the unifier to distinguish between repeated instances of identical nodes which should be recognized as distinct, and spurious sharing is eliminated.[18] Despite not implementing sophisticated substructure sharing, this is an advantage that the *agree* array storage unifier enjoys when unifying the same (top-level) argument TFS at multiple points in a single target,[19] and which any future design for array storage *sub*-structure sharing should be careful to preserve.

To further motivate departing from the per-node paradigm, I note that, in managed programming environments, node access via a system reference may not result in the direct access of memory anyway; it may be mediated through the additional indirection required to support automatic garbage collection.[20] Taken together, these factors suggest that independently centralizing the node storage for each top-level typed feature structure could result in improved performance in managed code systems, especially if combined with a synthetic access key that efficiently solves the feature arity problem. This is the motivation for array TFS storage.

The remainder of this chapter describes a system where all of the nodes belonging to a single top-level TFS are monolithically encapsulated. Because this engineering treatment departs from the formal treatments of typed feature structures reviewed in Chapter 2, it is useful to ex-

---

[18] The issue here is that some single TFS instance which is part of the grammar may be introduced at multiple points in the target structure during the course of a single unification, and the coreferences in each of these instances must not be conflated.

[19] With the ERG, the case happens most often with certain difference-list sub-types which can trigger well-formedness unifications at multiple points in the same structure during the course of a single operation. As noted earlier in this chapter, concurrent systems such as *agree* and CaLi (Callmeier 2001) trivially permit unfettered re-use of multiply-introduced structures.

[20] Many modern runtime environments implement Just-In-Time (JIT) compiling, where native code is prepared from the Intermediate Language (IL) on demand, and there is in fact no provision whatsoever for the direct interpretation of IL. In such systems, which includes the platform used by *agree*, certain types of memory access end up resembling the native memory access patterns.

plicitly clarify the terminology I will use. Where appropriate, notation concerning top-level structures from the functional formalism is adopted; for example, Q from (2.10) will refer to all nodes in a (top-level) TFS. To reiterate the issue raised in Footnote 7 on page 7, although the DAG conception of typed feature structures entails that each node has the formal form of another typed feature structure, in this thesis, "TFS"—typed feature structure—will be understood to refer only to structures whose topmost node is not referred to by any other TFS node. Excluded from the category are arbitrary portions of substructure within a TFS, namely any node $q_i : q_i \neq \bar{q}$. The term will refer to linguistic TFSes in the abstract as well as implemented array storage TFSes, but in both cases will designate only top-level structures.

The present work describes a system which abandons the fundamental assumption of per-node storage, namely, that each node necessarily has exactly one physical representation. A corollary, also not adopted, is that reentrancy, or coreferencing between nodes, manifests exclusively as *referential* equality. Referential equality means that two entities are taken to be the same if and only if their addresses are discovered to be the same, implying that they are physically co-extant. In the case of computer memory, nodes are construed as globally unique, and coreferencing is signaled by value equality amongst memory addresses—that is, a single memory address—which is reached via multiple paths. I raise this issue because, while preserving the conflation of *logical* nodes, the *physical* node storage model in array TFS storage does not incorporate the same reliance on implicit referential equality, and this difference becomes significant in the unifier implementation.

In array TFS storage, a single, contiguous allocation (per TFS) stores all of the structure's nodes (except the root node) as an array of 4-tuples. The main advantages of this scheme are that it is trivial to collectively discard the nodes of a TFS, and the overheads for $|Q| - 1$ runtime allocations,[21] per TFS, are avoided. Indeed, although implemented in a managed-execution environment, *agree* achieves performance comparable to a high-performance native parser.

2.3.1 Notation

This section summarizes notation from the relational algebra (Codd 1970) which is used to define array TFS storage and describe its operation. A relation R is an unordered set of zero or more homotypical n-tuples, or entities, $R =$

$$r : r = \langle p_0 : \tau_{p_0},\ p_1 : \tau_{p_1}, \ldots,\ p_n : \tau_{p_n} \rangle \tag{2.20}$$

Each n-tuple r associates a set of values $\tau_{r.p_j}$ with a set of named, typed properties, or attributes $\mathcal{P}_R = (\ p_0, p_1, \ldots, p_n\ )$. The property-value mapping is uniform across all tuples in a relation. One can think of R as a table of n rows $\{\ r_0, r_1, \ldots, r_n\ \}$ with column headings $\mathcal{P}_R$, with p_i giving the *name* of each property. Tuple properties are typed within some global domain of

[21] Recall from Carpenter's notation that Q is the set of all nodes in a TFS.

types $\tau \in \mathrm{T}$ particular to the application. A less cluttered method for indicating the types of tuple properties is

$$r = {}_{(p_0, p_1, \ldots, p_n)}\langle \tau_{p_0}, \tau_{p_1}, \ldots, \tau_{p_n} \rangle. \tag{2.21}$$

The tuples' values are referenced with 'dot' notation, so that $r.p_0$ is shorthand for the *value* of property p_0 for tuple r:

$$r.p_i = \tau_{r.p_i} = r_{(\ldots, p_i, \ldots)} \langle \ldots, \tau_{p_i}, \ldots \rangle. \tag{2.22}$$

This shortcut can be applied to relations as well—which entails removed abstraction, of course—so that $R.p_0$ refers to the *name* of property p_0 on relation R

$$R.p_i = \mathcal{P}_R\ (\ldots, p_1, \ldots). \tag{2.23}$$

Relations can be restricted by property or by range, with the π and σ operators, respectively. In the first case, the PROJECTION operator

$$\pi_{\mathcal{P}' = (p_i, p_j, \ldots)}(R) \tag{2.24}$$

expresses a new relation where each tuple from R has the values corresponding to the properties in $\mathcal{P}'$ deleted. This can result in duplicate tuple values, which are removed, since well-formed relations are, by definition, distinct. In some discussions, I relax the requirement that the range of a relation be distinct; these sections of the text will explicitly state the conditions of the suspension. Related to this is the need to retrieve a specific relation entity in tuple form, which is discouraged in relational algebra, where every operation is conceived as a total map from finite relation to finite relation (Tannen et al. 1992). For this, I permit referring to individual tuples via zero-based subscripting, meaning that r_i refers to some entity which is present in relation R. As relations are not ordered, further utility of this type of indexing is limited to cases when relation distinctness has been suspended and all of a relations tuples are known to be identical, or when the relation is known to contain a single *n*-tuple, and I wish to manipulate the singular distinct tuple. In either case, that tuple is—arbitrarily—any member of the relation, so it can be extracted from the relation as follows:

$$r = \mathrm{argany}(R). \tag{2.25}$$

Later in the presentation, in order to facilitate working with storage indexes, I incorporate an entity's storage index within its relation as a tuple property, but this will be understood as not changing the underlying storage of the relation. In practice, scalar values pertaining to the current node (such as a storage index) are explicitly propagated and carried for within the algorithm. This relaxed approach to indexing is possible because the array storage relations discussed in this thesis are read-only, and entity indexes, once established, never change.

Returning to a review of relational algebra operators, the SELECT operator $\sigma_{\varphi}(R)$ applies the propositional predicate φ to each tuple in $r \in R$ in turn, giving a new relation $S \subseteq R$ which contains zero or more matching tuples from R.

Finally, $Q = R \bowtie S$ describes the inner—or 'natural'—JOIN operation for relations, which, for each property in $\mathcal{P}_R \cap \mathcal{P}_S$, yields a new relation containing zero or more tuples Q with $\mathcal{P}_Q = \mathcal{P}_R \cup \mathcal{P}_S$,

$$Q = \{ {}_{\mathcal{P}_Q}\langle \dots \rangle_0, {}_{\mathcal{P}_Q}\langle \dots \rangle_1, \dots \} \tag{2.26}$$

according to

$$\sigma_{\bigwedge_{p \in \mathcal{P}_R \cap \mathcal{P}_S} r.p = s.p}(R \times S), \tag{2.27}$$

that is, a set of tuples of the form ${}_{\mathcal{P}_R \cap \mathcal{P}_S}\langle \dots \rangle$, selected from $R \times S$, where $r \in R$ and $s \in S$ share the same value for all corresponding property pairs, if any. It follows that, if $\mathcal{P}_R$ and $\mathcal{P}_s$ are disjoint, their inner join collapses to their Cartesian product

$$(\mathcal{P}_R \cap \mathcal{P}_S = \emptyset) \rightarrow (R \bowtie_{\emptyset} S = R \times S). \tag{2.28}$$

2.3.2 Formal description

I describe array storage with the relational algebra. At the outset, I point out that the objective is a theoretical model of a particular implemented system, as opposed to an idealized relational algebra model of the typed feature structure. Much of the vocabulary developed in this section is applicable to other designs, but the specific characteristics of the model presented in this section ultimately derive from engineering considerations—discussed in Chapter 3—and not from a criterion of descriptive elegance. The most egregious departure from an idealized model is that the *out*-tuple relation, taken alone, is not a well-formed (relational algebra) relation, a point which is the topic of Section 2.3.5.

I introduce the treatment by recalling the graph conception of a node. This is extended to a form suitable for relational algebra representation. As a storage model, this object model will at first contain no intrinsic provision for describing coreferenced nodes within the TFS. As the model is incrementally extended, this limitation is removed, and steps at which coreferencing is relevant will be pointed out. Notation from the functional formalism will be adapted where possible. However, Carpenter's use of π as a TFS path instance, σ as a type instance, and of $\bowtie$ as an equivalence relation, are in conflict with the relational algebra use of these symbols for the PROJECT, SELECT, and JOIN operations, respectively. To avoid ambiguity, I will designate paths with α and types with, e.g., t or $\bar{t}$.[22]

[22] TFS equivalence is not discussed in this thesis, so the symbol $\bowtie$, used in (2.40), refers to JOIN.

In unification grammar $G = \langle \textsf{FEAT}, \langle \textsf{TYPE}, \sqsubseteq \rangle, \mathit{Approp}(f,t), \dots \rangle$, a typed feature structure F consists of a set of nodes Q which each associate a type $t \in \textsf{TYPE}$ with zero or more feature/node tuples:

$$q_i = \langle\, t, \{\, \langle f, q_j \rangle, \dots \,\} \,\rangle\,, t \in \textsf{TYPE}, f \in \textsf{FEAT}, q \in Q. \tag{2.29}$$

One node, $\bar{q} \in Q$ is distinguished as the root of the TFS:

$$F = \langle \bar{q}, Q \rangle\,, \bar{q} \in Q. \tag{2.30}$$

Additional conditions apply to this definition, for example, that every $q \in Q$ be reachable from $\bar{q}$, that the structure contain no cycles, or that all of the features in $q_i = \langle\, t, \{\, \langle f, q_j \rangle, \dots \,\} \,\rangle$, if any, be appropriate for t. Other authors have thoroughly developed these points for the functional treatment (Carpenter 1992) and the graph treatment (King 1994).

Direct mapping of such structures as flattened relational algebra relations, while possible, is ill-considered as an engineering design. For example, because each node constrains a small fraction of the features in FEAT, a relational model with properties $\mathcal{P} = (\,\bar{t}, f_0, f_1, \dots, f_{|\textsf{FEAT}|-1})$ implies a vast and sparsely populated table of nodes. And although one could imagine distasteful workarounds, $\mathcal{P} = (t_0, t_1, \dots, t_{|\textsf{TYPE}|-1})$ is effectively ruled out because $\pi_{(t)}(F)$ provides no way to identify the recovered features. This thesis discusses unworkable schemes such as these no further. Rather than exposing contentful linguistic features from the grammar in a relational algebra, the focus of array TFS storage will be on schemes of the variety $\mathcal{P} = (\textsf{FEAT}, \textsf{TYPE}, \dots)$, that is, fixed-arity relational models into which the linguistic elements are subsumed as data (as opposed to relation properties).

The fundamental structural entity in array TFS storage is a 4-tuple, conceived as a pair of physically contiguous 2-tuples

$$a_i = \langle {}_{(\textsf{FEAT},\mathbb{Z})}\langle f, m_F \rangle, {}_{(\textsf{TYPE},\mathbb{Z})}\langle t, m_T \rangle \rangle, m_F \neq 0. \tag{2.31}$$

Each 2-tuple pairs an encoded linguistic feature or type, respectively, with an integer. The "mark" integers m_F and m_T will be discussed shortly. As a notational convenience, entities from relation a can be equivalently expressed in flattened 4-tuple form

$$a_i = \langle\, f, m_F, t, m_T \,\rangle. \tag{2.32}$$

In this thesis, a tuple of the form ${}_{(\textsf{FEAT},\mathbb{Z})}\langle f, m_F \rangle$ is called an *in-tuple*, and a tuple of the form ${}_{(\textsf{TYPE},\mathbb{Z})}\langle t, m_T \rangle$ (or ${}_{(\textsf{TYPE},\mathbb{Z})}\langle \bar{t}, m_{\bar{T}} \rangle$) is called an *out-tuple*.[23] In graph terms, the *in*-tuple corresponds to an arc, while the *out*-tuple corresponds to a node, but this presentation generally avoids the former term (except in the context of prior work) and more methodically develops the latter. Henceforth, the unwieldy property-typing prefix may be omitted from the 2-tuples,

[23] I will try to avoid using the term *edge* for describing the *out*-tuple, even though it is pervasive in the source code of the implemented system. Among other confusions, it has an unrelated sense in chart parsing.

since the tuple type can be trivially deduced. Relations of entity a (unordered collections of zero or more entities of the form ${}_{(\textsc{Feat},\mathbb{Z},\textsc{Type},\mathbb{Z})}\langle f, m_F, t, m_T \rangle$) are denoted by A.

Array storage for a given TFS consists of exactly one contiguous array of these tuples, given as the relation $\mathbb{A}$. Although a single array storage relation could, in principle, be used for all TFSes within the entire grammar,[24] designating the TFS as the extent of a storage relation is preferred, as it trivializes the problem of TFS discarding. The 4-tuples in $\mathbb{A}$ are unordered, which is to say the fixed order in which they happen to be arranged—and which happens to be of great interest to the unification algorithm presented in Section 4.4—is not formally significant.

In array storage, a typed feature structure F is defined by a root tuple $\bar{q}_F = {}_{(\textsc{Type},\mathbb{Z})}\langle t, m_T \rangle$ and a relation $\mathbb{A} \supseteq A$ on $\mathcal{P}_A = (f \in \textsc{Feat}, m_F \in \mathbb{Z}, t \in \textsc{Type}, m_T \in \mathbb{Z})$, $F = \langle \bar{q}_F, \mathbb{A} \rangle$. The definition will have more utility with its root tuple q decomposed:

$$F = \langle {}_{(\textsc{Type},\mathbb{Z})}\langle t, m_T \rangle, \mathbb{A} \rangle, \boldsymbol{\sigma}_{m_F=0}(\mathbb{A}) = \emptyset. \tag{2.33}$$

For convenient property access, the latter form is given flattened property names,

$$F = \langle t, m, \mathbb{A} \rangle. \tag{2.34}$$

For this thesis, this definition is complete, meaning that instances of F are given, and they encompass the entire storage and sufficiently describe all substructure of a TFS. When it is clear that only a single TFS is being discussed, reference to the relation $F.\mathbb{A}$ can be abbreviated $\mathbb{A}$. As additional descriptive vocabulary is developed, it is important to remember that only (2.33) most closely models the implementation representation. For example, consider the description of TFS node q developed later in this section:

$$q = \langle \langle t, m_T \rangle, A \rangle. \tag{2.35}$$

In this expression, the 4-tuple relation A is taken to represent the subset $A \subseteq \mathbb{A}$ of zero or more feature-value pairs directly associated with node q, which is not exactly a generalization of $\langle t, m, \mathbb{A} \rangle$, where the 4-tuple relation represents the unrestricted set of all nodes in a TFS. In fact, entities such as q—which encapsulate a (most probably) restricted 4-tuple relation—have no formal corollary in the storage implementation, wherein every operation ultimately involves only $F.t$, $F.m$, and $\mathbb{A}$. Only as fleeting memoranda in TFS manipulation algorithms—such as the array TFS unifier described in Section 4.4—do further derived entities find implementation purview. To summarize this point, I contrast a typed feature structure *definition*, $F = \langle t, m, \mathbb{A} \rangle$, with a *description* of its root node, $\bar{q}_F = \langle F.t, F.m, A \rangle$. The key task of the remainder of this section is to characterize this latter tuple, and, in particular, the (useful but theoretical) relation

[24] A related idea was investigated early in this research. In the earlier implementation, each feature was associated with a singleton array which comingled out-tuples from all (disjoint) TFSes. This "diffuse TFS storage" method did not offer an acceptably efficient solution to the TFS discarding problem.

A, for all remaining $q \in Q$, giving the complete set of *out*-tuples Q in F, the nodes of typed feature structure F.

The notation m denotes a *mark*, an arbitrary integer. In the discussion that follows, marks may be distinguished as *in*-marks or *out*-marks, depending on the context in which they appear. Typically the context is the type of tuple (feature *vs.* type, respectively) the mark came from. But mark values serve to relate, by inter-equality, paired 2-tuples within the same TFS as belonging to the same logical TFS node, so no mark value (except for zero, which is forbidden as an *in*-mark) is inherently context-bound. In particular, for mark m, the set of tuples

$$\pi_{f,t}\,\sigma_{m\neq 0\,\wedge\, m_F=m}(\mathbb{A}) \tag{2.36}$$

selects the set of attribute-value pairs $\langle f, t\rangle$ in feature structure F which belong to node

$$q_m = \langle \bar{t},\, \pi_{f,t}\,\sigma_{m\neq 0\,\wedge\, m_F=m}(\mathbb{A})\rangle. \tag{2.37}$$

The conjunction term excluding *in*-mark zero in the SELECT operation is redundant given the array TFS definition. From here on, it is omitted. This expression represents node reconstitution, as it permits the outward links of a traditionally-conceived node $\langle \bar{t}, \{\langle f, t\rangle, \ldots\}\rangle$ to be recovered. In prose, from every 4-tuple whose (non-zero) *in*-mark shares the same value as the mark of some governing *out*-tuple, a set of tuples $\langle f, t\rangle$ is extracted, and this node is of type $\bar{t}$, where $\bar{t}$ is the type stored in a "governing" *out*-tuple, discussed below. In the current design, for any out tuple $\langle t, m_T\rangle$ where type t has no appropriate features, m_T is required to have the value zero, a value which cannot be used to select further feature-value pairs. This is elaborated in Section 3.2.3.

The adequacy of array storage for linguistic modeling is achieved via two joining mechanisms.

(2.38) Each one-to-n equivalence between a non-zero *out*-mark and the $n \geq 1$ occurrences of *in*-mark m_F represents the set of n constraints on node q_{m_F}, or simply q_m. Nodes with a type that has no appropriate features are given $m_T = 0$, which explicitly selects no features. Observe that because m_F is barred from having the value zero, every constraint belongs to some *out*-tuple[25] which expresses that non-zero value as its *out*-mark.

(2.39) Every *in*-tuple is permanently bound (by storage adjacency) to exactly one *out*-tuple—although the latter may be value-equivalent with other occurrences in $\mathbb{A}$, giving a total of $n \geq 1$ copies. This is a form of n-to-one equivalence between *in*-mark m_F and *out*-mark m_T.

[25] More precisely, the correctness condition for any *in*-mark in array TFS A is that it be equal to either some non-zero *out*-mark present in $\mathbb{A}$ or equal to $A.m$, the mark of the root *out*-tuple: $m \in \pi_{m_F} A.\mathbb{A}$: $m \in \{A.m\} \cup \pi_{m_T}\sigma_{m_T\neq 0}(A.\mathbb{A})$.

a. If $n = 1$, then the one-to-one equivalence represents that q_{m_T} is a constraint for feature $q_{m_F}.f$ in q_{m_F}.
b. If $n > 1$, the many-to-one value equivalencies represent the n reentrancies from assorted node-feature pairs to coreferenced node q_{m_T}.

Although (2.37) appears similar to an inner join on the m property between the paired tuples in $\mathbb{A}$, this is not a precisely correct interpretation in relational algebra. Because array storage physically couples *in*- with *out*-tuples, the $\mathbb{A}_{OUT}$ relation cannot be considered an independent relation. In this form, it is not necessarily distinct. As noted in (2.39), coreferences in the modeled feature structure introduce a duplicate copy of ${}_{(\text{TYPE},\mathbb{Z})}\langle t, m_T\rangle$ for each reentrancy, an issue discussed further in Section 2.3.4. However, if we imagine for a moment an implementation where independent, well-formed relations $\mathbb{A}_{IN}$ and $\mathbb{A}_{OUT}$ each contain a non-zero property m, then node reconstitution is equivalently recast as a projection of a natural join

$$q_{m'} = \langle \bar{t}, \pi_{f,t}\boldsymbol{\sigma}_{m'=m}(\mathbb{A}_{OUT} \bowtie \mathbb{A}_{IN})\rangle. \tag{2.40}$$

Inner join is commutative, but here the *out*-relation is shown on the left to emphasize that it is the *out*-to-*in* join (2.38) that is explicit, whereas the *in*-to-*out* binding (2.39) is implicit in the storage. The coupled 4-tuple approach was chosen precisely because it avoids this extra join operation, and also for the simplicity of having a single garbage-collected allocation encompass the entire contentful storage of a typed feature structure. Additional engineering considerations are addressed in Chapter 3.

2.3.3 Node governance

The presentation of the node expression, above, elided discussion of the node's *governing out*-tuple, wherefrom the node's type $\bar{t}$ was recovered. This will now be addressed. In array TFS storage, the address of every node (except $\bar{q}$) fundamentally incorporates its immediate context with regard to some other node. At the outset, it is important to note that this context is not a path expression—from $\bar{q}$ to q, or otherwise. Rather, the context is a single node wherein the type and mark which selects q can be found. In traditional terms, the governing out-tuple corresponds to one specific parent node (out of the one or more) which has a path to q.

As an important first case, for the root node of a TFS, the governing *out*-tuple is given in the TFS definition. The governing *out*-tuple for TFS F is explicitly given as $\bar{q} = \langle F.t, F.m\rangle$, so the type of feature structure F is $F.t$. To reconstitute a node given its non-zero mark m, Equation (2.37) first gathers feature-value pairs according to $\boldsymbol{\sigma}_{m_F=m}(\mathbb{A})$. Taking this step alone, the set of 4-tuples corresponding to the feature-value pairs for $\bar{q}_F$, the root node of F, is given by

$$A_{\bar{q}_F} = \boldsymbol{\sigma}_{m_F=F.m}(\mathbb{A}). \tag{2.41}$$

Next, (2.37), repeated here (2.42), projects $\langle f, t\rangle$ pairs that include only the type associated with each feature. In practice, this is incomplete because it does not permit a way to recover further substructure. For this, I extend the node tuple notation given in (2.29) and (2.19) so that

it preserves the *out*-mark from its canonical 4-tuple, that is, the 4-tuple in which $\bar{t}$ was stored. It is straightforward to do this by including the entire governing *out*-tuple in the generalized node description, as shown in (2.43). For convenience, this tuple is given the flattened notation (2.44).

$$q_m = \langle \bar{t}, \pi_{f,t} \boldsymbol{\sigma}_{m \neq 0 \wedge m_F = m}(\mathbb{A}) \rangle. \tag{2.42}$$

$$q = \langle \langle t, m_T \rangle, A \rangle. \tag{2.43}$$

$$q = \langle t, m, A \rangle. \tag{2.44}$$

But as a reminder, note that this—and all other—extensions to the notation are developed for expository convenience, and do not change the underlying array storage definition of a typed feature structure F, which is still given by

$$F = \langle \langle \bar{t}, m_{\bar{T}} \rangle, \{ \langle \langle f, m_F \rangle, \langle t, m_T \rangle \rangle, \ldots \} \rangle. \tag{2.45}$$

From an engineering standpoint, it is worth understanding the abstraction captured by these extensions. Later in this chapter, I show that they mirror transient structures that facilitate array TFS manipulation. Having defined property names for the graph node tuple, I can now express the appropriateness condition which was elided in (2.29):

$$\bigvee_{q \in Q \cup \{\bar{q}\}} \bigvee_{f \in \pi_f(q.A)} : \mathcal{A}\mathit{pprop}(f, q.t)\downarrow \tag{2.46}$$

As this is a condition of formal correctness, it can also be viewed as a constraint on relation A_q, the set of rows in $\mathbb{A}$ corresponding to node q's feature-value pairs. It is not complete as such, however, since it does not assert, for example, that the set of features in A be distinct:

$$A: \left| \boldsymbol{\pi}_f A \right| = |A| \tag{2.47}$$

I have yet to describe the process of recovering the governing *out*-tuple from a given feature-value pair. This is required, for example, in order to ascertain the node's type. Here, the intricate structure of array TFS storage is more deeply revealed. Capitalizing on the property of a TFS that it is fully connected—that is—that all of its nodes are reachable from the root node $\bar{q}_F$, array storage associates the canonical storage location of node q with the one or more immediate parent *in*-tuples $\langle f, m_F \rangle$ that connect it to $\bar{q}_F$. More specifically, the governing *out*-tuple for node q is found amongst the set of 4-tuples

$$_{(\textsc{Feat},\mathbb{Z},\textsc{Type},\mathbb{Z})}S = \boldsymbol{\sigma}_{m_T = q.m}(\mathbb{A}), \tag{2.48}$$

that is, those with *out*-mark value equal to node q's *in*-mark. From this set, each 4-tuple $s = \langle f, m_F, t, m_T \rangle$ binds q, now equivalently

$$q = \langle s.t, s.m_T, \boldsymbol{\sigma}_{s.m_T \neq 0 \wedge m_F = s.m_T}(\mathbb{A}) \rangle, \tag{2.49}$$

as a value for feature $s.f$ in some node $q_{s.m_F}$.[26] Continuing now with the discussion of the governing *out*-tuple, observe that relation, $\pi_{t,m_T}(S)$, will always collapse to either zero elements (if and only if $q.m$ is zero), or exactly one element,

$$s = \text{argany}(\sigma_{m_T=m}(\mathbb{A})). \tag{2.50}$$

In the latter case, this gives the governing *out*-tuple for q, $\langle s.t, s.m\rangle$ or simply $\langle s.t, m\rangle$. This allows us to form a complete relational algebra expression for node q_m, the node corresponding to mark m:

$$q_m = \langle s.t, m, \sigma_{m_F=m}(\mathbb{A})\rangle. \tag{2.51}$$

However, this does not exemplify a useful operational mode in the actual array storage implementation. First, being physically bound into a 4-tuple, the *out*-tuple relation alone is not well-formed (details follow in Section 2.3.4), and value equivalence is not automatically recognized amongst its distinct entities. But furthermore, as I explain in Section 4.4.2, this is a problem for the unification implementation, which requires that each node be given a perfect mapping. It would not be immediately clear how to privilege one of the rows in S over the others. But more critically, array storage is not indexed in a way that permits efficiently selecting 4-tuples according to *out*-mark m_T, as expressed by S. The actual access method, which exploits the grammar's type appropriateness condition, is described in Section 2.3.6.

In array TFS storage, knowing value m permits retrieval of feature-value pairs for q_m, according to a match of m with zero or more *in*-marks, but given only m, it is not practical to discover $q_m.t$, the type of node q_m. Instead, based on a pragmatic observation, array TFS storage delegates to the caller the responsibility for maintaining $q_m.t$. The key observation is that navigation is implicit in every TFS operation; the requirement for examining a node never occurs in the absence of navigational context. When this appears not to be the case, there is always an outer operational scope which ultimately resolves to the root node of the TFS under consideration.

It is especially logical to delegate the responsibility for maintaining this information when considering the fact that the *out*-tuple governing q_m may be repeated, by value, across multiple rows in $\mathbb{A}$. Although each of these rows, if there is more than one, necessarily holds the same type value, the lack of a distinguished storage location poses a dilemma for the array storage unification algorithm presented later in this thesis. Section 4.4.2 describes how the conflation of coreferenced nodes, since it is not implicit in array storage, is instead effected by the unification procedure.

---

[26] The conjunction excluding zero is restored here as a reminder that the node we are studying—node q—is usually not realized as this sort of full-form tuple when the type of its governing *out*-tuple has no appropriate features. As will be discussed later in this section, tuples of the form $q = \langle\langle t, m_T\rangle, A\rangle$ model entities which most closely parallel fields which are only present in an operational, recursive, daughter stack frame. When a node has no features, it is likely that such a call is never made, in which case the frame will never exist.

Reviewing to this point, I first showed that the value of mark m pertaining to node q is required for selecting its features via $\boldsymbol{\sigma}_{m_F=m}(\mathbb{A})$. Next, we saw that the type of a node is stored in its governing *out*-tuple, but the resulting expression

$$q_m = \langle s_0.t, m, \boldsymbol{\sigma}_{m_F=m}(\mathbb{A})\rangle \tag{2.52}$$

still took as 'given' the mark value m that was sought. What I did not yet unambiguously illustrate is precisely where this mark comes from. Thus far, the presentation has been licensed by tacitly asserting a correspondence between m and the role of the node "pointer" in a traditional per-node TFS implementation. A practical implementation of any system will implicitly maintain references to entities that it is currently working with. In a per-node system, nodes are referenced by pointers whose values signify memory addresses. These addresses carry no inherent significance and require no further discussion. In contrast, the description of array storage is expanded such that the token used to reference a node incorporates domain knowledge. Therefore, it is instructive to examine more closely the runtime provenance and operational use of m.

We know that the first two fields in this tuple are notational extensions with undefined storage manifestation, but their form nevertheless provides a helpful insight. The mark m was in fact collapsed from the trivial identity

$$m = \text{argany}(\boldsymbol{\sigma}_{m_T=m}(\mathbb{A})).m_T, \tag{2.53}$$

applied to s. Indeed, taken as a single entity, the singular element of $\pi_{f,t}\sigma_{m_T=m}(\mathbb{A})$ is simply the full governing *out*-tuple. Accordingly, just as the node's governing type $q_m.t$ is not accessible given the value of m alone, neither is its governing mark $q_m.m$. Both of these fields—the governing type together with the governing mark—are stored as one or more copies of an identical *out*-tuple $\langle s_i.t, s_i.m\rangle$, none of which are accessible via the node's own mark.

It is now obvious that it is the responsibility of the storage consumer to pass not just the governing type, but rather the entire governing *out*-tuple into its recursive daughter frame. This burden is not onerous. All TFS operations—whether operating on traditional data structures or array storage—already pass a node pointer down to recursive stack frames, and this role in array storage is assumed by m. The difference with array TFS storage is that it may require the node type to be passed down as well. Acknowledging this returns us to the earlier description of a node, adjusted so that marks are not discarded, and where the node's type $\bar{t}$ is provided from without:

$$q_m = \langle \bar{t}, m, \boldsymbol{\sigma}_{m_F=m}(\mathbb{A})\rangle. \tag{2.54}$$

This concludes the core formal description of array TFS storage. The design suggests a new graphical form which is presented in the next section. Following this, Section 2.3.5 will briefly

explain an important practical implication of the bound 4-tuple design. To conclude the formal discussion, Section 2.3.6 will present an analysis of the operational retrieval mode.

2.3.4 Array TFS graphical form

Because array TFS storage disrupts the referential equality of joined nodes (by representing them with multiple disjoint, albeit value-identical, physical entities), the DAG interpretation of typed feature structures (presented in Section 2.1.3) could be taken as no longer applying to the new storage form. Instead, the form can be modeled according to a graphical model presented in this section.

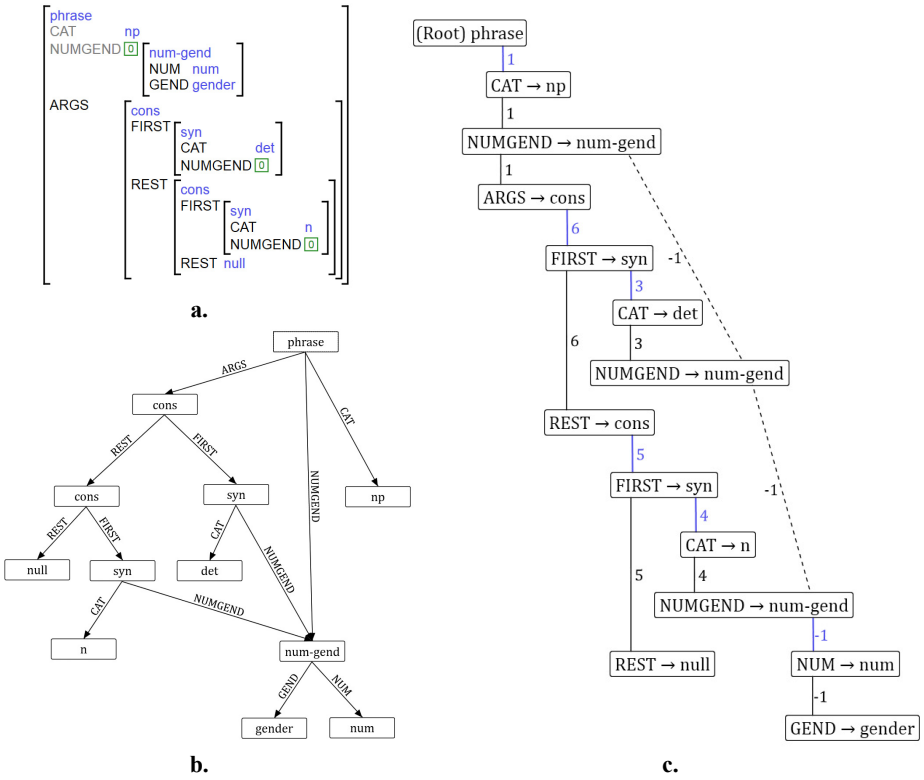


Figure 7. Typed feature structure (a.), shown with the traditional DAG interpretation (b.) and a new graph form which parallels the array TFS storage representation (c.).

Each "node" in (c.) corresponds to an array storage 4-tuple, a bound pair consisting of an *in*-tuple (shown as a connection point on the **left side** of a node) and an *out*-tuple (on the **right side**).

Connections are labeled with mark values, which relate *in*-tuples to *in*-tuples (left-to-left, black line), *out*-tuples to *in*-tuples (right-to-left, blue line), and coreferenced *out*-tuples (right-to-right, dotted black line).

Formally, two entity types are represented, the *in*-tuple (the feature "arc" in the traditional DAG) and the *out*-tuple, or "node," and the two appear to establish disjoint structures. However, in array storage each *in*-tuple is physically bound with exactly one *out*-tuple, complicating matters. One approach might be to model each on a graph of its own, with the graphs then interconnected. Instead, I fuse each tuple pair into a single "hybrid" node with exactly two connection points. Then, every unique mark value in the TFS (except zero) is represented by exactly one connected set of arcs which attach to these points. Connections to the *left side* of a node signify value equality with the node's *in*-mark, m_F, while connections to the right side of a node signify value equality with the node's *out*-mark, m_T. A special node for the root of the TFS therefore does not permit left-side connection.

The new form, a *non-directed*, acyclic graph, is illustrated in Figure 7. For clarity, the different types of connections are drawn differently. When an out-mark m_T connects to an in-mark m_F,

a blue line is drawn. This represents a node (out-tuple) selecting its immediate substructure (feature-value pairs) so the line always connects the right side of a node with the left side of another node. For example, the blue line labeled '6' indicates that this is the value of the *out*-mark in a node with type 'cons' that has feature-value pairs 'FIRST' and 'REST.' Note that although there are two feature-value pairs, only one of them—an arbitrary selection—is given this type of connection. This is because all connections are transitive in this non-directed graph, a point which nicely leads to a discussion of connections between feature-value pairs. These are shown with black lines, which connect amongst two or more *in*-tuples, and therefore always connect the left sides of nodes. Lastly, coreferencing—which relates a common mark value amongst two or more *out*-tuples—is shown with a dotted line connecting the right-sides of the corresponding nodes. In the example TFS in Figure 7, there is just a single coreference, which transitively relates arcs $m_F = -1$ and nodes $m_T = -1$, illustrating in this case all three types of connections.

This graphical representation will be useful for the array TFS unification examples in Section 4.4 because it emphasizes the particular internal structure (of the storage pattern) which is *co-opted* by the new algorithm. In particular, the discussion will note that any TFS unification result (except of course $\perp$) yields a structure which is guaranteed to be representable as a simple list of interconnections between structures which have the form illustrated here—namely, the original argument TFSes. Seen in this way, the unification procedure is greatly simplified. It becomes the task of revealing a structure which is *necessarily latent* amongst the given set of invariant argument graphs. The benefit of moving complexity out of the unifier—by allowing it select from amongst pre-made structures—is that parsing re-uses argument graphs across multiple unification operations, so work moved to other components may engender the amortization of the transferred costs.

2.3.5 Properness of the *out*-tuple relation

In Section 2.3.2, I mentioned that projections of the form

$$\pi_{t,m_T}(A) \tag{2.55}$$

may require special treatment in array storage. Because A binds *in*- and *out*-tuples together into a single 4-tuple, the intended identity of the *out*-tuples in S manifests as value—as opposed to referential—identity. In other words, in the current design, any $_{(\text{FEAT},\mathbb{Z},\text{TYPE},\mathbb{Z})}A$—and in fact $\mathbb{A}$ itself—resembles an inner join (2.40) where identical *out*-tuples are replicated by value. The effect is transparent to most consumers of array storage. Because *out*-tuples are delivered to them *by-value*, they have no way to discern any difference between the (identical) copies of a coreferenced *out*-tuple that originate from different rows in $\mathbb{A}$. The issue is of great concern to the array storage unifier, however. Because it uses a special access mechanism to obtain storage indexes, the lack of physical (as opposed to logical) node equality in array storage becomes patently clear. It becomes the responsibility of any client that "peeks behind the curtain"

like this to re-establish or somehow account for coreferencing that is not referentially "automatic." Fortunately, the unifier has at hand an ideal mechanism for this task—the scratch slot forwarding mechanism which is central to unification. The technique will be discussed throughout Section 4.4.

Singularity of the *out*-tuple relation

$$\pi_{t,m_T}\sigma_{m_T=m}(\mathbb{A}) \tag{2.56}$$

for all m in the context of the improper storage relation $\mathbb{A}_{OUT}$ insists that each distinct, non-zero value of m_T in $\mathbb{A}$ always be associated with the same type:

$$\forall m\left(m \in \pi_{m_T}(\mathbb{A})\right) \wedge (m \neq 0): \left|\pi_{t,m_T}\sigma_{m_T=m}(\mathbb{A})\right| = 1. \tag{2.57}$$

This is a required property of array storage which is not intrinsically entailed by the bound 4-tuple design. Accordingly, the requirement is transferred to the domain of operational enforcement, an acknowledged compromise attributed to well-motivated engineering considerations.

In the next section, the node description is further extended to more closely model the actual implementation, where selecting from $\mathbb{A}$ always requires both feature f and mark m to be supplied.

2.3.6 Feature enumeration

Although the preceding section serves as a minimally sufficient description of array TFS storage, additional important concerns reside in a zone which blurs the distinction between practical implementation and formal necessity. Efficient access being critical, a presentation of the precise storage and access methods will follow in Section 3.2. But first, it is important to reiterate the essential reliance that the current design establishes with respect to a particular aspect of the TFS formalism, namely, the feature appropriateness condition. This will be of particular interest to those interested in the application of the present work to other feature structure formalisms which do not adopt the condition.

Consider that, as presented thus far, array storage does not present a practical solution to the feature arity problem described in Section 2.2.3. An expression of the form

$$\sigma_{m_F=m}(\mathbb{A}), \tag{2.58}$$

while succinct for formal presentation, implies, for every node access, an exhaustive enumeration of every tuple in $\mathbb{A}$, clearly an unsupportable operational mode. As a reminder, in this example it does not suffice to stop the enumeration after finding the first match, because although the collection of *in*-tuples $\langle f, m_F\rangle$ taken alone is distinct, when considering only m_F there may be multiple satisfying tuples. This is as desired, of course, for these represent the zero or more feature-value tuples of a node.

A key point, however, is that the collection of *in*-tuples, taken as a whole, *is* distinct. In section 3.2.4, I explain that the implemented design hashes $\mathbb{A}$ on $\langle f, m_F \rangle$. For the current discussion, suffice it to note that $O(1)$ retrieval can only be obtained via expressions of the form

$$\sigma_{\lambda f \lambda m \ldots}(\mathbb{A}). \tag{2.59}$$

Considering this, it is apparent that we need to add feature *hints* to the node reconstitution expression developed previously.

Recall that the set of features appropriate for type t is an invariant property of the grammar, given by $\mathit{Approp}(f, t)$. From the start, the design of array storage has intentionally incorporated intrinsic, *as-given* reliance on this feature appropriateness condition, and here we see why. Without a short list of features to pair with a node's mark m—forming $\langle f, m_F \rangle$ tuples—and to then try, in turn, as hashable queries, array storage has no practical means of node retrieval. Naturally, that short list of features is the set of features that are appropriate for the node type.

From the previous section, we know that the *out*-tuple containing a node's type and mark—which, as an opaque entity, might be considered the array storage simulacrum of a node pointer—is explicitly passed down from a higher call frame. By this means, the node type is available for obtaining a list of appropriate features. Also at hand is the node's mark, for recovering the node's feature-value pairs. For convenience, give this externally-provided *out*-tuple $\langle q_m.t, q_m.m \rangle$ the alias $\langle \bar{t}, m \rangle$. We are now ready to give the general expression for array storage retrieval of node q_m:

$$q_m = \langle \bar{t}, m, \bigcup_{f' \in \textsc{Feat},\, f' : \mathit{Approp}(f', t)\downarrow} \sigma_{(f = f') \wedge (m_F = m)}(\mathbb{A}) \rangle. \tag{2.60}$$

With the exception of employing logical conjunction as a stand-in for $\langle f, m_F \rangle$ hashing, this expression now conveys a sufficiently accurate representation of the implemented storage method. Applying the transformations demonstrated in (2.41)-(2.44), we can express the feature-value pairs as relation $A \subseteq \mathbb{A}$:

$$q_m = \langle t, m, A \rangle. \tag{2.61}$$

Processing the feature-value pairs in $q.A$ is generally a task-specific concern which is peripheral to this exposition. A thorough example can be found in the presentation of array storage unification in Section 4.4. To conclude this section, then, it should be sufficient to note that the process described in this section can be continued with the zero or more *out*-tuples formed by $\langle a.t, a.m_T \rangle$, for each $a \in q.A$.

The original system design prohibited storing an *out*-tuple with type $\top$ (unless coreferenced, in which case it is has a negative mark value [Section 3.2.3]). The original intention of this design was to reduce the size of TFSes representing unexpanded type definitions, which tend to be sparsely constrained. With expanded TFSes from realistic grammars, actual space savings is

limited. Although array storage balks at storing *out*-tuple $\langle \top, 0 \rangle$, this is precisely the *out*-tuple which is returned for any $\langle f, m_F \rangle$ query for which no data is found. This is generally desirable behavior, as it automatically gives every node the appearance of a full set of appropriate features, albeit unconstrained. In reality, these identities are chimeric, for the unconstrained *out*-tuple $\langle \top, 0 \rangle$ is also happily returned for features which could never be appropriate for q. Array TFS storage strongly promotes feature appropriateness as the recommended retrieval vehicle, but in fact, array storage is agnostic to feature appropriateness. It falls to the caller to record *out*-tuples only for features which are appropriate for the node type; otherwise, later callers may have no efficient, systematic way to discover them.

In summary, although in principle the set of attribute-value pairs selected by a node can be recovered by exhaustively searching $\mathbb{A}$, this would be prohibitively inefficient. Appropriateness of f for t is a property of the grammar which—while technically extraneous to the array storage description—plays a critical role in a practical implementation. Accordingly, pairs are enumerated by querying only those features which are appropriate to a node's type. Although essential to efficient operation, the association is neither stored within nor referenced by the storage component and must be externally obtained.

2.4 Summary

This chapter began by reviewing established approaches to the formal description of typed feature structures, contrasting two quite dissimilar conceptions. Next, motivating considerations in the design of TFS storage systems were discussed. The chapter concluded with the formal description of a new mode of TFS storage, in which a single, independent monolithic tabular allocation holds all of the nodes for each "TFS," here taken to refer only to top-level structures which undergo (meta-)operational manipulation. Although the formal description borrowed notation from the relational algebra, the intended implemented form for an array storage TFS is a simple block of process memory. The details of an implementation of the scheme are provided in the next chapter.

3 Array TFS storage implementation

This section describes the implementation of the array TFS storage system which was formally described in the previous chapter. The method amounts to a specialized storage and retrieval abstraction over system memory. The demonstration harness, *agree*, is a new grammar engineering platform developed as part of this research. *agree* is implemented as an ECMA-335 Common Language Infrastructure (CLI)[27] application, and key aspects of the array TFS storage design were informed by engineering considerations related specific features and limitations of this environment; these will be discussed in this chapter.

Section 3.2 presents specific details of the array TFS storage implementation. Section 3.3 provides example representations of linguistically motivated TFSes stored in the new array form. Lastly and within the context of ongoing research, Section 3.4 describes an alternate to the hashing method currently used by the array TFS storage implementation; this section includes the results of preliminary analyses of the concept as it would manifest for the English Resource Grammar (Flickinger 2000). This approach, based on modulo division, shows potential as a faster, more compact solution to the feature arity problem.

3.1 Managed-code

While offering numerous advantages for rapid development of large-scale software projects, the "managed" CLI poses unique performance challenges. Automatic garbage-collection, a boon for design convenience, may require the introduction of a penalizing shim layer between every user-defined entity ("object") in the system and its physical representation. These extra headers consume a few bytes of extra space per object, but more importantly, may impose additional memory indirection each time an object is referenced.

The key observation is that engineering options are fewer—and thus extra care is required—when seeking the highest possible performance within managed programming environments. This was the experience of this project. Several storage schemes were investigated—and abandoned—before arriving at the configuration presented in this thesis. The CLI provides very few mechanisms for departing from its canonical paradigm; inevitably, the system design presented here is intrinsically tied to the specifics of the particular patterns, which are very limited in number, that enable extreme performance. This is not to say that the design of array TFS storage is fully implied by the constraints of its target platform, nor that any system which approximates the performance of the demonstration system within the CLI must be using the design described in this thesis. Rather, I am suggesting that when high performance is critical, there is a considerable reduction—relative to native-code programming—in the number of fundamental engineering abstractions that remain available for consideration.

[27] European Computer Manufacturer's Association (ECMA), Common Language Infrastructure (CLI), 5th Edition (December 2010). http://www.ecma-international.org/publications/standards/Ecma-335.htm

Most notably, in the CLI, the *array* is the only managed object that provides the requisite flexibility. CLI arrays are distinguished with first-class recognition as internal primitives with special properties. Dedicated instructions in the intermediate language (IL) provide index-based access to the (obligatorily homotypical) array elements, which are internally allocated as a contiguous block of memory. The number of elements in a CLI array (which can be zero), along with the single designation of their strong type, are permanently fixed when the array is created.

Were a CLI array limited to containing a contiguously allocated collection of (strongly-typed) references to garbage-collected objects, this thesis research could not exist in its current form. It would be extremely cumbersome to avoid per-node allocation and difficult to achieve adequate performance. Fortunately, the CLI platform allows certain types of managed array, under specific conditions, to be treated as simple variably-sized blocks of unsupervised virtual memory. The platform supports the declaration and instantiation of user-defined *value-types*, which circumvent the garbage collection mechanism (at the expense of referential equality) by everywhere exhibiting *by-value* semantics. Moreover, when used as array elements, the value type imposes an overlaid tuple structure on the underlying managed memory at no cost. This relatively obscure capability, without which an efficient managed implementation of array TFS storage would be impossible, is discussed in the next section.

3.1.1 Value types

In some managed languages, such as Java, arrays of user-defined types are always arrays of object *references*. Only in arrays of the built-in scalar primitives, such as integer, float, Boolean, etc., are the elements deployed by value *in situ*. Fortunately, C#—the CLI language used in this project—supports arrays of user-defined *value types*. A value type is a user-defined tuple prototype (or, when the term is used informally, an instance of the same) that manifests an altered semantics wherever it occurs or is manipulated. The mechanism is triggered by tagging the tuple declaration with the `struct`—as opposed to the `class`—language keyword (3.1). Value type instances receive a host of special treatments. The key semantic difference is that assigning a value type instance always causes its bitwise layout to be *copied*, a semantics referred to as *by-value*. Like the primitive scalars, they never share state with other instances and thus the notion of referential equality is undefined for these entities.[28] Additional examples of 'value type semantics' are that the type declarations for value types cannot express inheritance (although they can implement interfaces), and that their fields can be used without initialization (3.3) (although the runtime still zeros-out their memory for security reasons; see Section 4.4.3).

[28] As will noted in the next section, pointers can be used in C#, and thus a concept of referential equality—i.e. equality of physical location—can be foisted on value types. The end can also be rudely achieved via the 'StructLayoutAttribute/LayoutKind.Explicit' construct, which can permit (C-style) unions to be formed, but neither of these treatments imparts a notion of reference equality on the value types themselves. Curiously, there is such a notion in the CLI itself—so called 'managed pointers'—but the capability is not exposed in the C# language. One language that provides access to this mechanism is Microsoft's *C++/CLI*, a version of the C/C++ programming language which targets the CLI.

```
struct VT         // the struct keyword declares a value type                    (3.1)
{
    public int f;
    public int m_F;
    public int t;
    public int m_T;
}
VT vt;           // vt is a local instance of value type VT on the stack        (3.2)
vt.t = 3;        // fields of vt can be assigned without initialization         (3.3)
vt = new VT();   // this is one way to (re-)initialize an instance              (3.4)
```

Despite the vastly altered semantics, value type instances can be created in the same way as normal objects, by using the `new` keyword (3.4). This is an unfortunate conflation, because value type instances are essentially never[29] independently allocated within the garbage collected heap, which is what `new` implies to many programmers. It may help to think of a value type declaration as an overlay on some piece of memory. As a local variable or function argument, a value type instance is overlaid on a portion of the stack frame. As a member of a managed object, it is laid out, *in situ*, within a managed object. And as an array element, zero or more value type instances are laid out adjacently, in sequence.

To avoid excessive memory shuffling in a regime of by-value assignment, value types, when handled as standalone instances, are best indicated for cases of small tokens of no more than a couple dozen bytes. Significantly for the present work, however, declaring an *array* of value types does not devolve into an array of boxed value type references.[30] Rather, the declaration results in a true array, where a single, contiguous block of memory contains by-value images of each instance, adjacently and in sequence. This means that an array of value types subsumes a (relational algebra) relation, where individual fields of zero or more homotypical tuples are accessible by range and property name. Of course, being memory-resident constructs, these arrays enable capabilities beyond those defined for relational algebra tables, for example, the trivial, direct access of individual tuples via index values particular to the physical storage order.

Use as array elements licenses the declaration of value types larger than the practical limit mentioned above; in the role of an array element type, a value type is simply a property-naming template which is repeatedly overlaid on the array's memory, and its fields—rather than its contents as a whole—can be written to, and read from, where they lie. This use of value types as patterning templates incurs no operational cost at runtime, and largely serves as a convenience to the developer. However, in working within this system, the programmer must be keenly aware of the altered semantics, so as not to accidentally make any direct reference to the stored instances. Doing so invokes by-value semantics, of course, resulting in the (possibly oversized) tuple being (unintentionally) copied out of the array.

---

[29] *Boxing* is the exception; see footnote 30.

[30] Boxing allows a value type to be transported by reference (i.e., by persisting in the heap), but *by-value* semantics still apply: the "unboxed" value type, once recovered, shares no state with the source instance.

To illustrate, recall value type declaration (3.1) . Each instance of this tuple is 16 bytes, so the value type array `rgvt` (3.5) represents a single allocation of 16,000 bytes. Note that the managed array of value types itself is a normal garbage-collected object; the value type simply organizes its internal fine structure. Accordingly, each array storage TFS can be considered an independent, miniature heap which implements a specialized, domain-aware (because it capitalizes on the grammar's type-appropriateness condition) addressing scheme.

Naturally, the size of a managed array—which, once set, becomes permanently fixed—can be late-bound, at runtime. In (3.6) and (3.7), respectively, an integer is stored in the field f of the i^{th} tuple in `rgvt`, and then retrieved. This demonstrates direct, efficient access. On the other hand, (3.8) is not only inefficient, but probably incorrect with regard to the likely intention. By assigning the i^{th} array tuple to the local value type instance `vt`, the entire 16-byte entity is imaged into the local stack frame, where it no longer has any connection with its array source. Furthermore, the lack of referential equality means that assigning a value to field f will have no effect on the persistent array data. Only the local entity `vt`, which will expire with the stack frame, is altered.

```
VT[] rgvt = new VT[1000]; // a managed array of value types                    (3.5)
rgvt[i].f = 1;            // efficient in situ property access (writing)      (3.6)
int f = rgvt[i].f;        // (reading)                                        (3.7)
vt = rgvt[i]; // probably not intended: value types manifest by-value assignment
vt.f = 1;                                                                     (3.8)
```

Absent value type arrays, efficiently accomplishing the array storage scheme described in this thesis would require allocating each of the four properties of the 4-tuple relation as a separate array of primitive types. Each TFS would comprise four allocations instead of just one, and programming access in the language would be clumsier. In addition, the fragility of the overall system would increase by the requirement to operationally enforce tuple consistency across disjoint arrays. With the value type array, this is not a concern; obviously, if a tuple is a single, bound entity, it is impossible for it to become crossed with another.[31]

3.1.2 Direct access

The C# language also permits so-called *unsafe* access to primitive data, using C-style pointers, as shown in (3.9). Access to the entire memory block which comprises a value type array is also permitted. This means that, under controlled conditions, C# code can freely range over proscribed memory areas using native pointers. Strict limitations are enforced so that the guarantees of the managed environment—in particular, pertaining to type safety and garbage collection—are fully preserved. In a nutshell, the limitations amount to the prohibition of unmanaged access to managed *references*. There is also a mechanism, shown in (3.10), for accessing

[31] Of course, with concurrent editing by multiple threads—and with a value type whose total size is larger than the processor's native integer size—reading multiple fields of a value-type (which is) can still result in a "torn-read." This term describes the situation where one thread witnesses inter-field inconsistencies that arise due to non-atomic updating activities of another. In the current work, this possibility is ruled out by treating each TFS's storage array, once initialized, as read-only.

(permissible) fields directly *within* managed objects. The `fixed` keyword establishes a scope block within which direct manipulation of the entire array with C-style pointers is permitted, even though the array itself is a managed heap object.

```
VT* p = &vt;     // trivially, the address of a stack instance can be taken          (3.9)
p->t = 3;        // C-syntax is used for pointers
fixed (VT *pvt = rgvt)   // explicit pinning of the managed array from (3.5)          (3.10)
{
    pvt[10].f = 5;
}
```

A question is, why should this approach be used instead of the normal access mode illustrated in (3.7)—after all, value type array `rgvt` is itself a managed object? The answer is that (3.7) is actually a lighter-weight mechanism whereby the runtime environment automatically pins down the array for the duration of a single access.[32] To manually pin a fully-blown managed object—such as a CLI array—so that its internal fields can be directly accessed requires the explicit syntax of (3.10).[33] A summary of what is permitted when using pointers in C# is that, regardless of whether it points into the layout of a managed object, into a managed array, to a field within a value type, or to the stack, everything accessible to an unsafe pointer must include no managed object *references*.

Earlier in this research, I hoped that *agree* could be developed to adequate performance without appealing to the unsafe manipulation of value types. Eventually, I had to suspend this diktat for parts of the array storage system that were performance-critical. To retrieve a relation entity, the unsafe version pins 𝔸—yielding a pointer to its base address—and then maintains the pinning whilst navigating the hash tables, all using direct access and C-style pointer arithmetic. However, I found that even carefully tuned unsafe code that uses explicit pinning (3.10) is sometimes outperformed by managed array access, for which the environment itself can use lightweight, automatic pinning (3.7). For the most critical parts of the array storage system I experimented with hand-crafted `IL`[34] code, which can exploit the best of both worlds—automatic pinning and hand-crafted optimization—and this code does outperform canonical access.

Beyond array storage itself, the array storage unifier also heavily embraces speedy pointer-based access. Section 4.4.3 details how the unifier's scratch fields can be instantiated as an array of value types on the stack. This technique relies on another unsafe facility of C# that has not yet been mentioned. With restrictions, the language allows for stack-based arrays. With these so-called `stackalloc` allocations (3.12), an amount is subtracted from the current thread's

---

[32] The way this works is that the garbage collector will examine the stack, looking for special invisible local variables that are specially marked for pinning. Merely placing an object reference in one of these local variables instructs the garbage collector not to move the object. The condition can be proactively canceled (i.e. prior to the function returning) by storing a *null* reference in the special local variable. Otherwise, it becomes moot when the frame expires.

[33] In yet another pinning technique—not used in *agree*—a managed object can be explicitly pinned for an arbitrary duration, including across function boundaries.

[34] IL—intermediate language—refers to the synthetic intermediate byte code that is produced by a ECMA-335 compiler. The runtime environment's just-in-time compiler converts IL to native instructions upon first use.

stack pointer to reserve a region of unmanaged memory on the stack that can only be accessed with unsafe pointers.[35] Importantly for *agree's* purposes, the size of the stack allocation can be late-bound (determined at runtime); the point is illustrated by (3.11) and (3.12).

A value type can be used as the element type in `stackalloc` arrays. As noted, this technique—including its ability to accommodate a late-bound size—was tested in some versions of the array storage unifier presented in Section 4.4 of this thesis. As before, the value type specified as the type of the array element may not contain any object references.

```
int cb = 1000;                                                          (3.11)
VT* pvt = stackalloc VT[cb];                                            (3.12)
(*pvt).rgi[10] = 5;             // store an integer value in the 9th element   (3.13)
pvt->rgi[10] = 5;               // store an integer value in the 9th element   (3.14)
*(pvt->rgi + 10) = 5;           // pointer arithmetic can be used              (3.15)
```

Finally, with another use of the `fixed` keyword—a use unrelated to (3.10)—C# offers the even more obscure ability to instantiate an entire unmanaged array, by value, within a value type, as shown in (3.16). In this case, only the scalar primitives—and not user-defined value types—are permitted as the element type, and the number of elements may not be late-bound. These limitations limit the utility of the feature, but nevertheless, it was used in the *n*-way unifier (Section 4.3). Since the time that code was retired, *agree* no longer uses `fixed` unsafe buffers.

```
struct VT2 { public fixed int rgi[20]; } // 20 integers, in situ in a value type   (3.16)
```

3.2 Engineering

3.2.1 Root *out*-tuple

In array TFS storage, the root out-tuple of each TFS is not stored in the main storage relation $\mathbb{A}$. The decision to store the root *out*-tuple separately followed from the idea that $\bar{q}$ should not be bound to an *in*-tuple. This idea, in turn, was adopted because it enforces the notion that a TFS represented in array form does not represent the substructure of some other TFS. Since $\bar{q}$ is the (exactly) one node in A which is not paired with an *in*-tuple, it is impossible to specify a feature "arc" which leads to $\bar{q}$. Because each TFS A is essentially a private address space unto itself, substructure above its root is either an undefined notion, or entails a cycle.

A potential consideration might have been that deploying a distinguished *out*-tuple in A would result in an additional system-managed allocation, but because *out*-tuples are represented as value types, this is not an issue. As discussed in Section 3.1.1, the value type is laid-out *in situ* within A, so excess allocation is not a factor in the design decision. Ultimately, the separate-root design was chosen for the formal rigor it enforces, but the design did later present an unforeseen practical drawback, namely, that a set of unifier scratch fields (called a unifier scratch *slot*) for the root must always be set aside by the unification algorithm, a point that will be pre-

[35] I hesitate to describe these as 'arrays' here, because these stack-based allocations are unrelated to normal managed arrays, which always exist in the garbage-collected heap.

sented in Section 4.4.2. Of course, the adjustment always amounts to exactly one extra slot per TFS, which is easily arranged.

On the other hand, the design does degrade nicely when storing a TFS N whose root type t_N has no appropriate features. Modeling these structures is important because it is desirable, in some cases, to consistently assume that every type in the type hierarchy proffers an (expanded) feature structure which represents its constraints. The array form of N has distinguished root tuple $\bar{q} = \langle t_N, 0 \rangle$ and storage relation $\mathbb{A} = \emptyset$. An example is shown as Figure 8 on page 50.

It is not permitted for the root node of any TFS be coreferenced—since this would necessarily introduce a cycle—so the *out*-mark value stored in the root node is normally a positive—in practice, the value 1. Currently, the only exception to the root *out*-mark having the value 1 is for the array TFS storage instances which represent the expanded constraint for a type with no appropriate features. In this case, the number of 4-tuple rows is zero, and the TFS consists of just a root *out*-tuple, which will then have a mark value of zero.

3.2.2 4-tuple layout

This section briefly describes the engineering implementation of the 4-tuple relation

$$a_i = \langle f, m_F, t, m_T \rangle. \tag{3.17}$$

In the current implementation, each property is a 32-bit integer, so each tuple is 16 bytes wide.[36] This means that a typical TFS of around 600 nodes occupies around 10,000 bytes of memory, all told. The sets of features FEAT and types TYPE are fixed aspects of the grammar, so individual features and types are selected by integers which are permanently assigned when the grammar loads.

Using a 4 byte integer for each member of the 4-tuple is wasteful. In particular, the use of 32 bits for identifying a feature means that realistic grammars use less than one millionth of one percent of the available range. Revision 10342 of the ERG uses only 206 distinct features, so using 16-bit integers for $a.f$ would pose no overflow danger.

As for the mark values, with the English Resource Grammar, ample headroom is still available with $a.m_F$ and $a.m_T$ at 16 bits. Since each TFS has its own domain of marks, this would limit each TFS to around 60,000 nodes. A brief analysis follows. Although there is little reason to disable rule daughter (ARGS) deletion during parsing—except perhaps for debugging—this limit can be approached in that artificial scenario. I suspend for a moment the question of whether such a large structure could be produced in a reasonable amount of time. Still assuming no ARGS deletion, a binary branching grammar, and an average rule TFS size of around 500 nodes, the length of a sentence corresponding to a TFS with 60,000 nodes is about

[36] Rather than introduce additional notation to distinguish feature- and type-identifier integers from features and types themselves (as conceived within the grammar), this thesis uses f and t, respectively, for both. Where the distinction is not immaterial to the presentation, it will be noted.

$$2^{\left(\log_2\left(\frac{60000}{500}+1\right)-\log_2 2\right)/\log_2 2} \approx 60 \text{ (words).} \tag{3.18}$$

Unrelated considerations (namely, the inherent complexity of parsing) render the analysis of sentences of this length (with realistic grammars) intractable, so, for the time being, 60,000 nodes is an acceptable limit for the number of nodes in a single TFS.

In *agree*, the type integer $a.t$ incorporates the most intricate structure. As stored in array TFS storage, the type identifier value is stored in the low-order twenty-seven bits. The remaining bits—the upper five bit positions—are flag bits whose functions are summarized in Table 1. The top type ⊤ in ⟨TYPE, ⊑⟩ receives special treatment. For all grammars, it is always assigned a type identifier of zero. The flag bits are polarized such that a value of zero for the integer as a whole reflects the correct state for non-coreferenced ⊤.

purpose	high 5 bits	description	ERG count
coref	`1___0`	Coreferenced. Bit is set if the TFS node has more than one path leading to it.	n/a
leaf	`_1__0`	Leaf. Bit is set if the type is a leaf type. This is used to optimize type unification.	3,003
appf	`__1_0`	Bit is set if the type has appropriate features. Used to optimize TFS traversal, including by the unifier.	5,915
type	`___00`	Non-string type	8,019
string	`___10`	Special string type.	1
string value	`_1010`	The low 27 bits indicate a string from the grammar's table of strings. String identifiers are assigned when the grammar loads.	46,832
skolem	`_1011`	The low bits indicate a Skolem constant used in tactical realization.	~30

Table 1. High-order bit flags encoded in type identifier $\boldsymbol{a.t}$. The low 27 bits indicate a type from the grammar's type hierarchy or a string identifier. Each type is assigned an identifying integer when the grammar loads. The identifier values are assigned according to a topological order on ⟨TYPE, ⊑⟩ such that $\boldsymbol{t_0 \sqsubset t_1 \rightarrow (id)t_0 < (id)t_1}$, a property that is heavily exploited by the unifier implementation, which gives it an asymmetrical form. The order enables the trivial (and obligatory) alignment of the more subsumed type (putatively—since this is prior to the actual type unification calculation) with a designated unifier function call position. Several code paths within the unifier are eliminated by entailment. For example, in the latest code (where the unifier never conceives new structure, and instead describes the result structure via scalar forwarding between input structures alone) is streamlined when it can explicitly refer to the input which will ultimately be chosen as the forwarding representative (*viz.*, the more derived type, or the greater-valued type identifier).

The high bit of $a.t$ indicates a coreferenced node. For historical reasons, this information is redundant with the *out*-tuple's mark being negative. In fact, coreferencing is indicated by the same bit position (the most significant bit) in both $a.t$ and $a.m_T$. It is an error for these to ever be inconsistent in any *out*-tuple.

Bit 30 always accurately indicates whether the type has any appropriate features. Because array storage mandates close coordination with the grammar's appropriateness condition, this flag is a powerful optimization used in every graph traversal. It is pervasively referenced throughout *agree*.

Summarizing the overall bit consumption of array storage entity a, $a.t$ legitimately needs at least twenty bits, so it would be reasonable to continue to use a 32-bit integer. Combining this with the analysis of the other fields in the 4-tuple, we see that the current width of 16 bytes could comfortably be recast with only ten bytes:

$$f\ (16) + m_F\ (16) + t\ (32) + m_T(16) = 80 \text{ bits} = 10 \text{ bytes.} \tag{3.19}$$

Unfortunately, a record size of ten bytes is not too alignment-friendly. Whether the memory savings outweigh the penalties for accessing unaligned fields is a question for further empirical study.

3.2.3 Mark assignment

As an operational convenience, a negative integer is assigned as the *out*-mark for any node $\langle t, m_j \rangle$ where m_j has more than one appearance as an *in*-mark. This indicates that $\langle t, m_j \rangle$ is a coreferenced node, that is, a node which can be reached by more than one other node. Coreferenced nodes select their attribute-value pairs in the same way as non-coreferenced nodes, according to equality of their *out*-mark with a set of zero or more *in*-marks.

Therefore, negative mark values always have more than one occurrence amongst the *in*-tuples relation $\mathbb{A}_{IN}$. This is the set of reentrancies to that coreferenced node. Every *out*-tuple in the array TFS which has a particular negative value as its *out*-mark must share the same type value t. If this requirement—or if the requirement that there be more than one such node—is violated, the array TFS is corrupt, and this signals not any linguistic condition, but a programming error.

As for non-coreferenced nodes: if a type has no appropriate features—a condition indicated by a special bit in the type-identifier integer—its *out*-mark will always be zero. It follows that any mark value greater than zero is guaranteed to occur in only a single *out*-tuple. That node's feature-value pairs are found by iteratively pairing the mark with each of t's appropriate features and querying with each resulting key. Conceptually, the *out*-mark matches with zero or more *in*-tuples, which identify the node's substructure: its feature-value pairs. Coreferenced nodes work the same way, if they have appropriate features. This is illustrated with an example in Figure 11 in Section 3.3. $\top$ nodes do not need to be explicitly stored unless coreferenced, because the storage system returns $\top$ in reply to any failed query.[37] For coreferenced $\top$ nodes—or in fact any logically coreferenced node, regardless of whether its type has appropriate fea-

[37] Historically in *agree*, attempting to store $\langle \top, 0 \rangle$ was flagged as a fatal error. In recent work, this aspect of the design has been altered, with the ultimate result that the unifier's fallback fetch mechanism, described in section 4.4.8, also became unnecessary. The minimum required condition was to ensure that the canonical expanded TFS for every type in the grammar manifest *root coverage*, meaning that its storage layout necessarily include a row for every feature appropriate to its root type (at a minimum), regardless of whether the feature is constrained. When combined with the formalism's well-formedness requirement, doing so guarantees that any result TFS can be described by a patchwork formed exclusively from the input arguments' storage layouts. Because the unifier—which co-opts these layouts—can always elect to incorporate the well-formedness TFS before unifying the actual argument nodes, this assures the unifier that a full complement of scratch slots will be available for any type it could possibly encounter.

tures—a negative *out*-mark must be stored so as to record the logical equivalence of the multiple storage representations.

To achieve the highest performance, formal correctness of array storage is asserted only in the context of a cooperative contract between the storage system itself and the storage consumer. An example was already given wherein the canonical access paradigm necessitates reference to the grammar's feature-appropriateness condition, which is entirely outside the purview of the storage component. Additional conditions must be operationally observed by the storage consumer. First, marks must be assigned such that $\mathbb{A}_{IN}$ is distinct, in order to prevent the unintentional conflation of nodes. *Out*-tuple m_j necessarily recurs when $m_j < 0$, but all occurrences must consistently record the same type t. So a second condition is that no TFS may contain out tuples $\langle t, m \rangle$ and $\langle t', m \rangle$ where $t \neq t'$.

It is also clear that, within the same 4-tuple, if the *in*-mark equals the *out*-mark, a cycle is described. Although the DELPH-IN joint reference formalism does not permit cycles, there is nothing inherent in array TFS storage which precludes them. Therefore the detection and prohibition of cycles is not the responsibility of the array storage system.[38] Finally, in any *out*-tuple where t has no appropriate features, m_j will be zero, and this, in turn, entails that no *in*-tuple may ever exhibit $m_i = 0$.

3.2.4 Hash access

Access is the heart of array storage, and is obviously the most performance-critical aspect of *agree*. For $O(1)$ access, the 4-tuples in $\mathbb{A}$ are hash-indexed by *in*-tuple $\langle f, m_F \rangle$, which is a range-complete projection of $\mathbb{A}$. In other words, the set of *in*-tuples in $\mathbb{A}$ is distinct:

$$\left| \underset{f,\, m_F}{\pi} \mathbb{A} \right| = |\mathbb{A}| \tag{3.20}$$

Abetted by a feature-appropriateness lookup, this hash key form permits the complete set of attribute-value pairs corresponding to a particular *in*-mark to be quickly recovered, in accordance with (2.60). The array storage scheme is itself agnostic about feature appropriateness; it will gracefully return *out*-tuple $\langle \top, 0 \rangle$ for any $\langle f, m_T \rangle$ tuple for which no constraint is stored.

Hashed fetching of a single 4-tuple a according to its *in*-tuple $\langle a.f, a.m_F \rangle$—supplemented with the ability to return a's storage index (later given as $a.ix$)—is the only operational mode supported by array storage, and it is sufficient for all of *agree's* grammar processing tasks, including expansion of the type hierarchy, unification, parsing, and generation.

It bears repeating that array storage does not maintain any index over a TFS's *out*-tuples, so the node type governing a given mark is not readily retrieved. As discussed at the end of Section 2.3.2, such a design delegates to the querying party the responsibility for enumerating only the

[38] Only tiny cycles such as the one mentioned here are detected "for free" by the storage system. Therefore, a comprehensive approach to cycle detection, if needed, may properly lie beyond the scope of node storage and retrieval.

appropriate features for a node, and this further implies knowledge of the node's type. In practice, this is never a problem, since—to prime the process—the root type of the TFS is explicitly available (it is explicitly stored, separate from $\mathbb{A}$), and from that point on, it is natural and trivial for code that recursively traverses a TFS to privately maintain the information needed to build the hash keys of interest.

Hashing requires that each entity $a_i \in \mathbb{A}$ be augmented with a 'next' field, which implements a singly-linked list of tuples whose hashes collide. In the implementation evaluated here, this is a managed CLI array ${}_{\mathbb{Z}}H$ of scalars ${}_{(\mathbb{Z})}\langle h \rangle$ which is allocated adjunct to $\mathbb{A}$ such that $\left| {}_{\mathbb{Z}}H \right| = |\mathbb{A}|$. The inelegance of this disjoint storage approach was justified by a few factors. Chiefly, controlled evaluation of the 5-tuple plan against the 1/4 disjoint plan was decisive in favor of the latter, a result that I attribute to better alignment. If each field in 4-tuple a is 32-bits, or four bytes, then the 4-tuple is 16 bytes in total, which is obviously an alignment-friendly figure. On the other hand, packed alignment of 5-tuples (20 bytes each) causes every alternate storage relation to be misaligned for 64-bit (8 byte) access.

Because *n*-way unification (Section 4.3) is no longer the primary unification engine in *agree*, a second factor which influenced allocating the 'next' field as an adjunct array is no longer relevant. The issue is nevertheless interesting. The *n*-way procedure is able to determine when each node in the result TFS becomes definitive, so it makes sense for that algorithm to directly produce each nearly-complete a record at that time. However, because unification may fail later on, it is not wise to go so far as to allocate the final CLI array. In any case, *n*-way unification does not know the required size for such an array until its single pass completes. Instead the a records are imaged to *n*-way's topmost stack frame. Before starting *n*-way unification, the method calculates the worst-case maximum number of tuples that could possibly occur in the result TFS, and obtains an incubation buffer of this size. It is this estimation procedure that influenced the decision to keep the array storage tuple a compact: the estimate is always in vast excess. The estimate is then multiplied by the byte size of each entity to get the size of the stack allocation.

In addition to associating a 'next' field with each entity, each TFS has a dedicated table of initial hashes, ${}_{\mathbb{Z}}\mathbb{H}$. As with H, this is a relation of entities ${}_{(\mathbb{Z})}\langle h \rangle$. $\mathbb{H}$ is used to abstract the physical storage order from the hashing mechanism, which allows the unifier to privately prepare the CLI array $\mathbb{A}$, adding records in arbitrary order and without concern for hash collisions. It then transfers the completed array to array storage, which trivially (i.e., by reference) installs it in the TFS. There is no opportunity in this procedure to rearrange records so that they are stored at their initial hash index—or added to the collision bucket of the record that is already there. $\mathbb{H}$ provides the mechanism for each tuple $a_i \in \mathbb{A}$ to be hashed regardless of the physical array index at which it happens to be stored.

Within this basic setup, I evaluated the performance of several hash function variations. The results are instructive, possibly illustrating complex interactions between orthogonal factors. For example, further research along these lines might seek to isolate differences which are due to a specific set of feature- and type-identifier value assignments. A related research program is described in Section 3.4. Also requiring further investigation are issues of which hash function yields the most efficient IL instruction sequences—or which of these sequences is then rendered most efficiently by the 64-bit JIT compiler. As one of the very few parts of the system which lies *below* the unifier, the storage hash described here is performance critical, so these issues merit close scrutiny.

Table 2 gives results for the best-performing of the hash functions that were considered across multiple scenarios. The function that has performed the best, regardless of $|\mathbb{A}|$, uses a table of $2^8 = 256$ initial hashes and the trivial hash function

$$(2^8 - 1)\ \&\ (f + m_F). \tag{3.21}$$

What is surprising about this result is that, parsing with the English Resource Grammar (Flickinger 2000), TFSes are produced in a size range of about 400-1200 nodes. With only 256 initial hashes, the bucket load is high, which can translate into up to five expensive list traversal operations per query. The explanation must be that the processor instruction for loading a byte from a 32-bit integer—which replaces, for example, modulo division for clipping the computed value down to the size of the hash table—is much faster than doing modulo division with an arbitrary divisor, and even measurably faster than the bitwise masking used when the table size is a power of two that is *not* divisible by eight.

clip function $\boldsymbol{h} =$	**index size**	$\boldsymbol{k} =$			
		$\boldsymbol{f + m_F}$	$\boldsymbol{f * m_F}$	$\boldsymbol{(m_F \ll 8) + f}$	$\boldsymbol{(f \ll 8) + m_F}$
k % $npp(\lvert\mathbb{A}\rvert)$	next pseudoprime ≥ $\lvert\mathbb{A}\rvert$	40.7	42.1	43.1	44.0
k % $\lvert\mathbb{A}\rvert$	$\lvert\mathbb{A}\rvert$	40.6	43.0	43.0	44.8
k & $(2^8 - 1)$	256	**38.2**	41.3	(f) 38.9	(m_F) 38.6
k & $(2^{10} - 1)$	1024	39.9	40.9	40.0	39.7

Table 2. Evaluation of array storage hash functions. Four runtime hash functions—which create a seed $\boldsymbol{k}$ by adding, multiplying, or shifting the feature identifier and *in*-mark—are examined with four clipping methods—which truncate $\boldsymbol{k}$ down to the size of the index table by performing modulo division (%) or bit masking (&). Results report the total parse time, in seconds, for 287 sentences in the 'hike' corpus.

Although it is easy to understand that multiplication is more expensive than addition, it is surprising that bit-shifting and masking are relatively expensive also, so much so that well-intentioned strategies that aim for better hash distribution by integrating all available input entropy may actually be better off just ignoring the lower-entropy signal (f) altogether. Recently, a possible explanation for this has come to my attention. Since 2001, PC microprocessors have included a set of specialized instruction-set extensions (known as SSE2) which facilitate, among other things, high-performance bitwise operations. It is possible, given the inestimable value of each square micron of real estate on today's CPUs, that the availability of a high-

performance alternative on the same silicon justified compromises in the performance of the traditional x64 bitwise instructions. Furthermore, it is known that the current version of the .NET x64 JIT compiler does not emit SSE2 bit operations. Taken together, these considerations could explain the poor performance of the bitwise tests in Table 2.

3.3 Example

This section provides examples of typed feature structures as represented according to the array storage scheme described in this thesis. As described in Section 2.3.2, an array TFS storage entity A binds a table of 4-tuples $\mathbb{A}$ with a single 2-tuple $\bar{q} = \langle t, m_T \rangle$, which serves as the root entry point. For logical clarity in the depictions that follow, the root *out*-tuple will be adjoined to the top of the table depicting $\mathbb{A}$, but this does not imply that the root *out*-tuple is stored in $\mathbb{A}$; as noted in Section 3.2, it is stored separately. Tables representing array TFS storage relations are always arranged with the *in*-tuples ("arcs") on the left and *out*-tuples ("nodes") on the right. The distinguished root tuple, being an *out*-tuple, is thus shifted to the right side.

The examples will not show data related to the hash access mechanism described in Section 3.2.4, since this aspect of the design employs well-known techniques. I also will not label the rows of $\mathbb{A}$ with index values, since storage order is irrelevant to the canonical access mode outlined in Section 2.3.6. The unifier, however, does examine row ordering, so index values will be shown alongside the array rows in the unification examples in Section 4.5.

The first example shows the trivial case of a TFS which represents the expanded constraint for a type which has no features. Details were provided in Section 3.2.1 and the case is illustrated in Figure 8.

[agr]

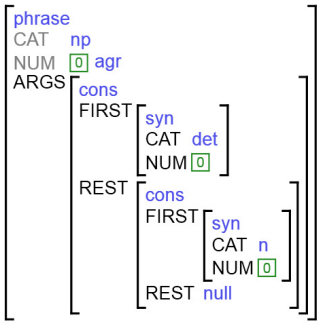
in-tuple		*out*-tuple	
m_F	f	t	m_T
	ROOT→	agr	0

Figure 8. Every type in the type hierarchy must have an expanded TFS, even if it has no appropriate features. The requirement that $\boldsymbol{m_T}$ be set to zero for all non-coreferenced nodes whose type has no appropriate features applies equally to the root *out*-tuple.

Next, consider the NP rule from the demonstration grammar used throughout this thesis. The array TFS storage representation is shown in Figure 9. The example illustrates coreferencing in array TFS storage. Coreferenced nodes are always indicated by a negative mark value; in this TFS, there is a single coreferenced node, associated with mark value -1. Interpreted as an *out*-mark, any negative value must be consistently paired with the same type across all *out*-tuples in $\mathbb{A}$ that shares that mark value; accordingly, the tuple $\langle \text{agr}, -1 \rangle$ is repeated three times. This is the *by-value* equivalence which was the topic of Section 2.3.4. Contrast this with the DAG representation of Figure 5, where coreferencing is implemented according to *referential* equality, that is, via an assumption of global node uniqueness which is implicit in the model.[39] The contrast is also evident from the fact that

[39] Here, I am referring to physical—not logical—node uniqueness; the latter is of course a non-negotiable correctness requirement of the linguistic formalism. Physical uniqueness means that a node is unique across the virtual memory of the process. Array TFS storage abandons this condition while preserving logical uniqueness; in the DAG view, each logical node has a unique physical representation.

there are ten nodes (and eleven arcs) depicted in Figure 5, whereas there are eleven rows in the array storage representation (that is, not including the root tuple) of the same TFS in Figure 9. In fact, the number of rows in array storage relation 𝔸—and thus by extension, in any array storage TFS—will always equal the number of DAG arcs, and not the number of DAG nodes.



$NP \rightarrow DET\ N$

in-tuple		*out*-tuple	
m_F	f	t	m_T
	ROOT→	phrase	1
1	CAT	np	0
1	NUM	agr	-1
1	ARGS	cons	5
5	FIRST	syn	2
5	REST	cons	4
2	CAT	det	0
2	NUM	agr	-1
4	FIRST	syn	3
4	REST	null	0
3	CAT	n	0
3	NUM	agr	-1

Figure 9 Array TFS storage. The typed feature structure for the grammar rule shown above is represented as a single root *out*-tuple, plus a relation of paired 2-tuples 𝔸 which has eleven (additional) rows. Compare to Figure 5, where the same structure is depicted as a directed graph.

This suggests an intuitive view of array storage as a *feature-centric* storage paradigm (whereas DAG conceptions could be considered node-centric). Although this tabular, feature-centric approach removes the convenience of "automatic" referential equality between nodes—which in turn requires the unifier to take steps which reinstate the condition—it turns out that the model simplifies the unifier itself, overall. This conclusion is the topic of Chapter 4, but in a nutshell, when unifier scratch fields are available on a *per-arc* (as opposed to *per-node*) basis, a new guarantee arises, namely, that the unification result structure, if it exists, can always be represented as a simple interconnection of pre-existing structures, resulting in no (additional) manipulation of operationally-variant structures.

Returning now to the discussion of Figure 9, it is clear that, for coreferenced nodes, there is no storage row in 𝔸 that might be taken as the canonical *physical* storage location for logical node $\langle \text{agr}, -1 \rangle$; amongst the three rows that express this *out*-tuple, there is nothing that obviously distinguishes one of them relative to the others. Logical node identity is still preserved, though, because the same *out*-tuple is retrieved no matter which of these rows is queried. Furthermore, since this *out*-tuple will always contain the same (negative) value of m_T, its substructure is correctly conflated. Details on this point are given next.

The coreferenced node $\langle \text{agr}, -1 \rangle$ in Figure 9 does not itself have any substructure. To illustrate such a case, the modeling of syntactic agreement in the demonstration grammar is expanded to incorporate the notion of linguistic gender. The new type and its subtypes are shown Figure 10. Together with the number distinction from earlier, it is deployed in a new structure 'num-gend' intended to model agreement. This allows us to consider the TFS shown in Figure 11, which

expands upon the example shown above by illustrating the use of negative mark values within *in*-tuples.

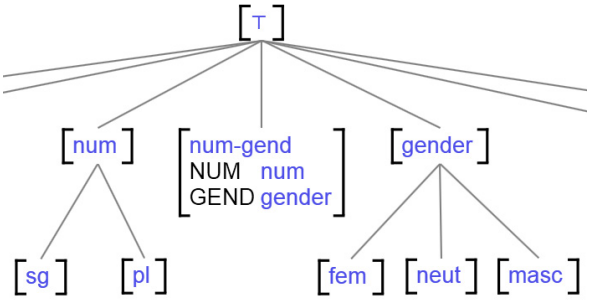


Figure 10. Some unconstrained types are added to the type hierarchy of the simple grammar in order to illustrate the occurrence of substructure below a coreferenced node.

In fact, any integer value—whether negative or positive—receives equivalent treatment when interpreted as an *in*-mark m_F by the *in*-tuple hashing system. The one consideration regarding m_F values is that zero is never permitted. This is to avoid confusion with the use of zero as the *out*-mark value m_T in any out-tuple whose type has no appropriate features (and which is also not coreferenced). Setting $m_T = 0$ for such cases is obligatory in the array storage implementation presented here. Therefore, although the fact that a given type has no appropriate features is operationally obtained from the grammar (which immediately short-circuits further queries), forbidding $m_F = 0$ prevents the accidental association of random substructure (*viz*., the substructure it would otherwise select) with (all of the) non-coreferenced leaf nodes in the TFS.

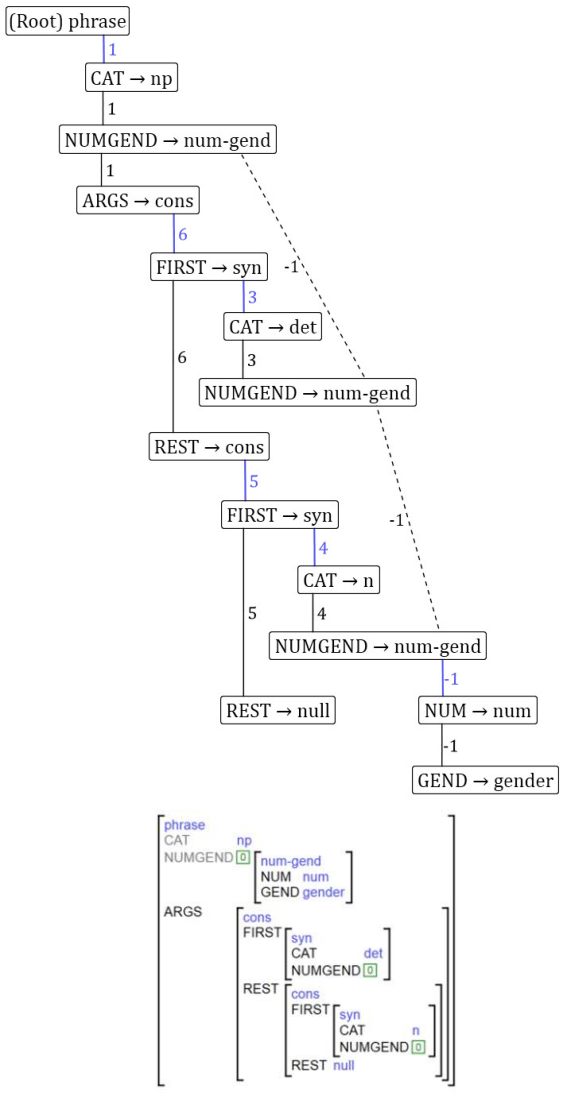
In the examples shown thus far, the feature which selects the coreferenced node happens to always be the same, but this is not necessarily always the case. In fact, as evinced by the fact that the set of *in*-tuples is necessarily distinct across any given storage relation $\mathbb{A}$, the repetition of feature 'NUMGEND' does not imply any relationship between the tuples that contain it. Rather, those tuples refer to arcs which are *logically* distinct. In other words, while a logical *node* may have diffuse physical representation in array TFS storage, this is not the case for the TFS's feature-value pairs, where each logical arc has exactly one physical correlate.

Note that I was careful not to state that *in*-tuple distinctness applies "globally," because in fact the requirement applies only within a single array TFS storage entity. Stated more generally, mark values obtained from a given TFS $A = \langle t, m, \mathbb{A} \rangle$ have no meaning outside the context of $\mathbb{A}$, the storage relation from which they were obtained.[40] This summarily interdicts the sharing of substructure between TFSes A and B in the current design, but several solutions can be envisioned. Most straightforwardly, any TFS-relative datum, such as a reference to some portion of internal substructure, can be made globally unique by simply pairing it with (a reference to) the TFS it pertains to, and then externally carrying the resulting tuple.

Another solution would be to have the unifier issue mark integers from a global sequence counter as it emits new structures. In the current implementation, the unifier starts over with an independent sequence for each result TFS, a design chosen in order to reduce contention in concurrent operation. Using a global sequence is a viable alternative, providing mark values

[40] Exceptions are the special out-mark values, 0 and 1, discussed earlier in this section and in Section 3.2.3.

are stored with adequate bit-width in $\mathbb{A}$. The issue I hint at here is sequence overflow during lengthy batch processing sessions. Naturally, the occurrence of integer overflow would defeat the primary purpose, which is to guarantee unique addressing. To combat the problem, a mid dle-ground solution would be to use independent mark sequences on a per-parser-input basis.



in-tuple		*out*-tuple	
m_F	f	t	m_T
	ROOT→	phrase	1
1	CAT	np	0
1	NUMGEND	num-gend	-1
1	ARGS	cons	6
6	FIRST	syn	3
6	REST	cons	5
3	CAT	det	0
3	NUMGEND	num-gend	-1
5	FIRST	syn	4
5	REST	null	0
4	CAT	n	0
4	NUMGEND	num-gend	-1
-1	NUM	num	0
-1	GEND	gender	0

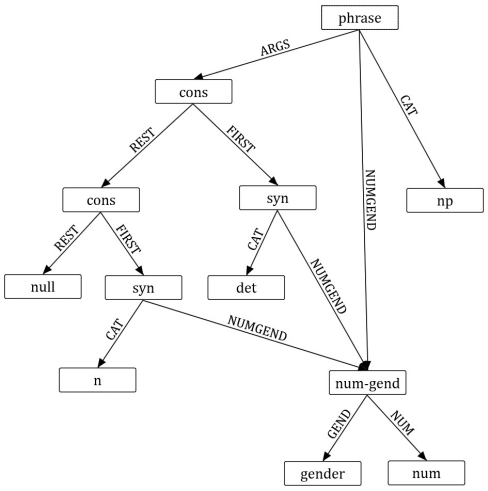


Figure 11. Because the *out*-mark $\boldsymbol{m_T}$ obtained from the multiple instances of a given coreferenced *out*-tuple always has the same value, substructure of the modeled node is uniquely identified. This is because, as an *in*-mark, a negative value selects nodes in exactly the same way as the (positive-valued) mark of a non-coreferenced node. Compare the table shown here to Figure 9, which had no negative values in the *in*-tuple column $\boldsymbol{m_F}$. This shows that the problem of recovering a physical node mapping from array TFS storage has a local solution, that is, the re-correlation of duplicated *out*-tuples can always be resolved within a single layer of structural nesting.

In other words, each sentence parsed or semantics generated would have a private sequence counter, with respect to which mark values would be unique. Such a solution would also mitigate contention to a small degree. The detailed exploration of proposals such as these are left for future work.

This concludes the presentation of examples of the representation of array storage TFSes. The examples in this section focused on the persisted representation of fixed storage instances. Further examples, given in Section 4.5, show how static structures such as these are manipulated via an active process, TFS unification, that provides a principled foundation for the modeling of linguistic structure.

3.4 Future work in array storage

The access paradigm used in *agree's* array storage is an area of continued research. In the current design, and as noted in Section 2.3.2, the ordering of the 4-tuples in array storage relation $\mathbb{A}$ is not operationally significant. Related to this, a research avenue I am currently pursuing is to abandon hashing and order the 4-tuples in $\mathbb{A}$ according to primary key m_F and secondary key f. This would permit a single 4-tuple pointer to be handed off to the unifier. From here, the unifier would itself walk forward to directly examine all of the 4-tuples for a node, thereby eliminating $n - 1$ array storage queries for each node with n feature-value pairs. For a variation on the method, each *in*-mark value m_F might be assigned to be the index, in $\mathbb{A}$, of its first feature-value pair.

Specifically, the research involves a variation on array storage that even further exploits the feature appropriateness condition. Because the 206 features used by the English Resource Grammar always appear in one of only 120 different configurations, the latter can be taken as fixed in a study of the feature arity problem (Section 2.2.3). In other words, assuming grammars exhibit the property that reentrancies in the type hierarchy $\langle \mathsf{TYPE}, \sqsubseteq \rangle$ are, on average, moderate in number and relatively distant from the root $\top$, arity conflicts between consistent collections of sets of co-occurring features will be limited, and an expensive arity analysis phase during grammar startup is justified.

I extend the work of Callmeier (2001, 50), who proposes three schemes for pre-computing fixed feature layouts. The methods achieve a neutral result in Callmeier's evaluation. A key point, however, is that this work is constrained by the fact that the Tomabechi implementation uses *in situ* scratch slots, and so must generally be able—during unification—to adapt to dynamic changes in the set of features appropriate for a node. Once a feature arity map is selected for a node, it is inefficient to switch to another, so Callmeier studies methods for minimizing these 'coercions.'

I instead note that, with detached scratch slots and array storage, it is not necessary to predict nodes' feature arity unification dynamics. It is sufficient to pre-compute only the limited, global set of actual appropriate feature configurations—and not consider the (set of) all of the ways in which they can be consistently combined. This is because, with the array systems described in this thesis, a TFS is written only once, during the second pass of the unifier, at which time

each node's exact feature configuration is definitively known, and at which time it is trivial to select the pre-computed arity map for the node's type. Thereafter, the TFS is read-only.

Therefore, the feature arity encoding problem is both simplified and pushed down into the storage domain. The requirements are now to find the most efficient way to model a graph which associates (type) values around a limited, known set of arity (feature) configurations. Efficiency here entails that both the space consumption and runtime fetching be minimized. In this early stage of the research, I have identified a simple and promising approach.

A *feature configuration* $\mathcal{W}$ is an (unordered) set of features that co-occur in some TFS node.

$$\mathcal{W} = \{f : f \in \textsf{FEAT}\} \tag{3.22}$$

As noted, with its 206 features, the ERG manifests only 120 distinct feature configurations, and the well-formedness requirement ensures that this figure is a globally-invariant upper bound for the grammar. Recall from 3.2 that, when a grammar is loaded, each feature $f \in \textsf{FEAT}$ is given a unique identifying integer. Let all possible mappings be designated $\mathcal{K}$ so that a particular mapping is $\mathcal{k} \in \mathcal{K}$.

Given some $\mathcal{k}$, for each possible feature configuration $\mathcal{W}$ there exists an optimal encoding modulus $\hat{u}_{\mathcal{W}}$. This is the integer which, when used as divisor for each feature identifier $f \in \mathcal{W}$, the set of remainder values which spans a minimal range is given:

$$\hat{u}_{\mathcal{W}} = \underset{u:\{1\ldots(\max_{\mathcal{W}} f - \min_{\mathcal{W}} f\}}{\operatorname{argmin}} \left(\max_{\mathcal{W}}(f \,\%\, u) - \min_{\mathcal{W}} f \,\%\, u\right) \tag{3.23}$$

The optimal modulus $\hat{u}$ is an invariant property of each feature configuration—given some $\mathcal{k}$—and each type in $\langle \textsf{TYPE}, \sqsubseteq \rangle$ is associated with exactly feature configuration, its set of appropriate features.

Because the feature-to-integer mapping is produced at the discretion of the implementation, it is possible to optimize (3.23) by seeking the mapping of integer values to features $\hat{\mathcal{k}}$ for which the maximum optimal modulus $\max_{\mathcal{W}} \hat{u}_{\mathcal{W}_i}$ amongst all feature configurations in the grammar is minimized:

$$\hat{\mathcal{k}} = \underset{\mathcal{k} \in \mathcal{K}}{\operatorname{argmin}} \left(\max_{\mathcal{W}} \hat{u}_{\mathcal{W}_i}\right) \tag{3.24}$$

A concern of the modulus encoding scheme for TFS storage, to be described shortly, is that it could be wasteful of memory. (3.24) identifies the feature-to-integer assignment which, when used in the scheme described below, results in a minimum of wasted memory. Unfortunately, the number of possible mappings $|\mathcal{K}|$ is very large, and (3.24) is not tractably computed. Nevertheless, experiments with the ERG show that total wasted memory using the method would not be excessive.

Now detailing the modulus encoding array TFS storage approach, the feature-value pairs, (if any) for node $q = {}_{(TYPE,\mathbb{Z})}\langle t, m\rangle$ are stored sequentially in the rows of a relation $\mathbb{B} = \{b\colon {}_{(\mathrm{TYPE},\mathbb{Z})}\langle t, m\rangle\}$, beginning at the zero-based row index given by

$$m + (f \,\%\, \hat{u}_{\mathcal{W}_t}). \tag{3.25}$$

There is no hash table and no feature identifier is stored. Mark value m—now in a use unrelated to the current array storage design—indicates the starting physical index in $\mathbb{B}$ of the first in a set of rows. The number of rows reserved for a node is known to equal the optimal modulus $\hat{\mathrm{u}}_{\mathcal{W}_\mathrm{t}}$ of the feature configuration $\mathcal{W}_t$ associated with the node's type t. Within the set of contiguous slots (that is, relative to m), the node or nodes associated with each feature appropriate to t occupy an offset which is the result of the modulus division.

As with array TFS storage, it is the responsibility of the clients of the storage to supply a feature identifier as part of the address for the node they wish to retrieve—or walk the sequential entries in the storage relation directly themselves. Naturally, this information is available from in the grammar, and it can be managed in whatever manner befits the different storage uses and activities. The variation in client uses, however, poses more of a problem for this method than it does for ordinary (i.e. non-modulus) array TFS storage. The next paragraph examines this issue.

In modulus feature encoding, each encoding may contain holes, which represent wasted space. Attempting to retrieve one of these slots would represent the error of asking for the value of a feature which is inappropriate for the type. Recall that in the ordinary array TFS storage method contributed by this thesis, requesting the value of an inappropriate feature silently returns the successful result that there is no constraint. Lacking a hash table, however, the lighter-weight modulus scheme described here would not inherently exhibit this graceful fallback. Although the modulus division for some errant feature requests might happen to land on an empty slot, most would wrap around to the index of the value associated with some other unrelated feature. Because inappropriate requests return random data, it is more critical with the modulus method to establish external measures for ensuring that values are only requested for appropriate features. To maximize performance, internal subsystems which are known to request only appropriate features, such as the unifier, might be allowed to bypass this safeguard, while storage activities that are less controlled, such as user interaction, maintain the checks.

The main savings of modulus feature encoding is that it does not require feature identifiers f to be stored, immediately cutting the storage requirements for every TFS. To evaluate this method, I analyzed feature the configurations of the ERG. The 120 feature configurations exhibit between zero and fourteen features each, but for these experiments, the diagnostic feature 'RNAME' was excluded, so $\max_{\mathcal{W}}|W_i| = 13$. Table 4 shows the results of these experiments, comparing the minimum modulus computation using unaltered feature identifier assignments—where feature identifiers are assigned in the order they are encountered while loading

the grammar—and comparing them to a mapping $\mathcal{k}$ which represents the lowest value for $\max_{\mathcal{W}} \hat{u}_{\mathcal{W}_i}$ found so far in a program of exploring equation (3.24). [41]

	actual number of features per node	un-optimized $\mathcal{k}$		optimized $\mathcal{k}$	
		$\hat{u}_{\mathcal{W}_i}$	excess features per node	$\hat{u}_{\mathcal{W}_i}$	excess features per node
min	0.00	0.00	0.00	0.00	0.00
average	5.65	7.76	2.11	7.25	1.60
weighted avg	6.47	7.27	0.80	7.59	1.12
max	13.00	25.00	15.00	16.00	2.00

Table 3. Analysis of the 120 feature configurations in the ERG for optimal-modulus feature arity encoding.

	ERG total nodes	Waste
actual requirement	53,412	-
un-optimized $\mathcal{k}$	60,010	12.35%
optimized $\mathcal{k}$	62,589	17.18%

Table 4. Analysis of the total number of TFS (substructure) nodes stored for all expanded type definitions in the ERG. The actual requirement is compared to the proposed modulus encoding scheme using two different feature-identifier mappings.

The results are instructive; the mapping which should be space-optimized does indeed waste less when all of the 120 feature configurations are given equal weight, but when the results are adjusted to model the total number of nodes consumed by the loaded grammar, the optimized mapping actually wastes 5% more space than the un-optimized method. This means that a disproportionately large fraction of the nodes that comprise the loaded ERG use the smaller feature configurations, and that the presumably random mapping is lucky enough to be less wasteful for these smaller configurations than the hard-earned "optimal" configuration.

This analysis is preliminary. In particular, additional work is needed to characterize the use distribution of the feature configurations, and to then reformulate objective function (3.24) in terms of this distribution. Future work should seek a mapping $\mathcal{k}$ that minimizes waste in the smaller configurations at the expense of fitting the larger ones (which are much more difficult to solve anyway). From these results I am able to conclude that further investigation is justified, because it appears that the total amount of memory used in this scheme—waste plus non-waste—will not exceed the memory requirement of the (non-modulus) array storage method proposed in this thesis. The latter stores a feature identifier for every arc, which the proposed scheme does not, and the tests indicate that number of wasted array slots in the proposed method will not be egregious.[42]

This section gave a preview of my ongoing work in array TFS storage research. An alternative to the main method of this thesis was sketched and the method promises to reduce the memory

[41] For these experiments, a brute-force search was used to explore (3.24). Solutions with $\min_{\mathcal{W}} \hat{u}_{\mathcal{W}_i} = 17$ can be found relatively easily, but only a single solution with modulus 16 was obtained in several hours of computation.

[42] Note also that each 2-tuple in the proposed scheme is half the size (or less) of a standard array TFS 4-tuple.

footprint compared to the current array TFS storage design. The method is a considerable departure from the implemented system, and only the preliminary feasibility experiments presented here—and no prototype—have been completed. It is important to mention that the evaluation and results presented in Chapter 5 pertain not to this sketch, but rather the array storage system described as the main contribution of this thesis.

3.5 Summary

This concludes the discussion of array TFS storage. The chapter began by contrasting an axiomatized formal conception (Carpenter 1992) with graphical treatments (King 1989, 1994) which more closely parallel engineering practice. The bulk of the discussion was a formal presentation, in notation adapted from relational algebra, of the array storage scheme proposed in this thesis. In this method, all of a TFS's nodes are stored within a single monolithic allocation. Engineering motivations for this approach were presented, including the observation that such a system, by reducing pressure on the system garbage collector, may be especially suited to use in managed-runtime environments. Details particular to the *agree* implementation of array TFS storage were also discussed.

To establish the efficacy of array TFS storage in realistic linguistic application, it was deployed in the *agree* grammar engineering platform. The next chapter discusses the unification algorithms investigated as part of that system. The chapter begins with a review of prior work in algorithms for linguistic unification, including a description of *n*-way unification, a novel and interesting (but now superseded) method developed as part of this research. The latest unification algorithm developed for *agree* is presented in detail. The method is implemented over array TFS storage, but aspects that are applicable to traditional TFS storage are noted.

4 Unification

Given the descriptions of two objects, *unification* is defined as the computation of the most general object which satisfies both descriptions, if such an object exists. Research into determining this *most general unifier* began in the context of term unification, the unification of variables in mathematical equations (Herbrand 1930). A thorough review of early work in term unification can be found in Knight (1989).

In linguistic applications, grammatical objects are encoded as typed feature structures, and their interactions are modeled via unification, the lone operation applied for this purpose. A total binary operator $\mathcal{F} \times \mathcal{F} \rightarrow \{\bot\} \cup \mathcal{F}$,[43] feature structure unification produces the single most general TFS which contains all of the information from two input TFSes, if such a structure exists.

Chapter 4 has the following structure. The first section provides an overview of well-formed unification. The well-formedness requirement is an instructive point of departure because it generally demands, as a fundamental design requirement, that the unifier support reentrancy, which in turn can interfere with certain unification algorithms and efficiency techniques that have been proposed in the prior literature.

Modern unifier designs draw from a rich set of research literature. Section 4.2 reviews this work, focusing attention on the milestones most relevant to the present work. For example, particular attention will be paid to van Lohuizen's work on concurrent unification, since the *agree* unifier presented in this thesis is also thread-safe, supporting the operation of the *agree* concurrent chart parser and generator components.

Section 4.3 describes *n*-way unification, a previous research direction that received a complete implementation in *agree*, but which has since been deprecated in favor of the array TFS unifier. Although the efficiency of the *n*-way implementation was hampered by uninteresting technical details, the method itself explores a certain theoretical bound on unifier complexity. The *n*-way algorithm also resists categorization by an informally-defined model that has focused the research discussion since the late 1980s, suggesting that the *n*-way method is fairly unusual in comparison to the other reviewed work. Specifically, as an implemented expression of a theoretical lower bound on unifier work that nevertheless shows acceptable—but not stellar—performance, the *n*-way implementation invites renewed examination of those traditional categories. Two questions that emerge are, first, whether classifying algorithms according to variously interpreted descriptions of "copying" behavior is a technique which the research community has outgrown, and second, whether the described classes remain relevant to contemporary implementations.

[43] Note that well-formed TFS unification is not a partial function $\mathcal{F} \times \mathcal{F} \nrightarrow \mathcal{F}$, since $\bot$ is an interpretable result.

Following the discussion of *n*-way unification, Section 4.4, presents the description of array TFS unification. This, along with the array TFS storage that it relies on, is the primary contribution of this thesis. The section contains formal discussion as well as details on implementation. The final two sections of this chapter give examples of the operation of the array TFS unifier, and provide offer summarizing comments, respectively.

4.1 Well-formed unification

It is important at the outset of this chapter on unification to distinguish feature structure unification—the subject of this chapter—from type unification, upon which it depends. The two are similar in that they both produce either a most general mutually-consistent result, or an indication that no such result exists. Type unification, however, appeals only to the type hierarchy $\langle \textsc{Type}, \sqsubseteq \rangle$ for determining the consistency between simple scalar types. Recall that, upon loading, $\langle \textsc{Type}, \sqsubseteq \rangle$ was automatically modified so that every such outcome is deterministic.

Typed feature structure unification, on the other hand, is the process of producing a new TFS: the most general TFS that is consistent with a set of input TFSes. This is the role of a unification algorithm, or a TFS *unifier*. In TFS unification, *type unification* is the operation applied between every pair of isomorphically coincident nodes—to be precise, between the *types* of the pair—in order to form the result structure.

Section 2.1.5 described well-formedness for typed feature structures. Recall that this means that at no time may any feature structure be inconsistent with any constraint authored as part of the grammar. In the DELPH-IN joint reference formalism, every TFS is required to be well-formed at all times. After initially making all of the entries in the grammar well-formed, the burden, during parsing and generation, of maintaining well-formedness is reduced, but not eliminated. Because the unifier produces new structures during parsing or generation, it assumes the responsibility for ensuring that those structures are well-formed.

During parsing, unification can successfully combine structures in ways that produce results that are not automatically well-formed, even though all of the input arguments were well-formed. If these feature structures cannot be made well-formed—that is, if the additional unifications required to make them consistent with the relevant parts of the grammar, fail—then the original unification is taken to have failed. For the unifier, this is a condition that must be evaluated at every node. The condition that signals that the additional step must be taken is that the type unification between two nodes results in a type which is not identical to either of the input node types. When this is detected, the unifier must temporarily cease its pairwise traversal of the input arguments to conduct an additional unification. It must fetch the TFS representing the expanded constraint for the type in question, and unify this entire structure at the target node in the new structure before resuming with the original work.

With simple unifier designs, the interruption is easily accommodated with a recursive call to the unification function. The matter is complicated by issues discussed throughout this thesis:

- If shared structures containing coreferences are introduced into the same result structure more than once, their coreferences must not be conflated.
- If a reference TFS is used in a recursive call, and if it contains in situ scratch slots which are in use, then alternate measures must be available
- A concurrent unifier must permit reentrancy with regard to its scratch slot mechanism.

In short, well-formed unification is a formal requirement that affects nearly all areas of unifier design. Research into linguistic computational unification has been active for three decades and a wide variety of avenues have been explored; many of these ideas inform the solutions used in today's increasingly sophisticated algorithms. A chronological review of published work in linguistic unification is the topic of the next section.

4.2 Prior work in linguistic unification

Early work in unification grew from research on term unification in the Prolog logic programming language. In term unification, each of the input arguments must have identical property sets in order for unification to succeed. Later applications, especially in linguistics, began to adopt the more flexible graph unification paradigm, a generalization of term unification which allows the combination of nodes with disjoint feature sets. As the body of literature in linguistic unification grew, researchers in computational linguistics established descriptive terminology that has been generally adopted to describe three desiderata for an efficient unification procedure:

- Unification should avoid **early copying**. Wroblewski (1987) proffered this term as describing wholesale copying of input arguments prior to the start of a destructive unification operation, but later researchers commonly adopt Tomabechi's (1992) stricter interpretation, which also counts all output structures produced during any unification that eventually fails.
- Unification should avoid **over copying**. Also introduced by Wrolblewski, this refers to creation of any structure which does not end up becoming part of the actual result.
- Unification should minimize **redundant copying**. Kogure (1990) points out that a result TFS often includes sub-structures that are identical to structures that exist elsewhere, and these, ideally, do not need to be copied either. In other words, structure sharing should be maximized across the entire system.
- **Lazy copying** (Godden 1990, Tomuro and Lytinen 1997) variously refers to schemes which attempt to defer copying activities until they are proven absolutely necessary.

For broad analysis, these are useful distinctions, but their inherent blurriness has not previously been emphasized. For example, the over-copying definition penalizes the production of "result" structure, but not excessive manipulation of temporary structures. The difference between the two is often not clear-cut, since at a fundamental level, both simply involve storing values in memory. Consider hyper-active parsing (Oepen et al, 2005), where temporary structures are

retained, unrealized, for participation in further unifications. Should these count as "result structure?"

Furthermore, although the categories implicitly reward two-pass unifiers for exhibiting no over-copying, intensive scratch slot activity during the first pass could erode this advantage. A more meaningful measure would consider the average memory bus throughput, per unification. Further pursuit of these types of experiments is left for future work; for the present thesis, the definitions are adopted with requisite caution.

Next, I review the research chronology in algorithms for linguistic unification. Where appropriate, connections to array storage unification are highlighted. Several research threads have been pursued in four decades of work, all aimed at reducing the computational burden of graph unification. For completeness, research avenues that were not further pursued are also placed in context with brief summaries. Two broad classifications to note at the outset are the classification of methods as single- or two-pass, and the contrast between destructive and non-destructive techniques. Single-pass methods are obviously simpler than two-pass methods, but the latter are preferred when an inexpensive first pass can detect failures. Destructive methods consume at least one of the input graphs in producing the result structure, and I begin with a description of the seminal algorithm which falls into this category (Section 4.2.1). Linguistic applications require non-destructive methods, such as those reviewed in Sections 4.2.2 through 4.2.10.

4.2.1 UNION-FIND

The first unification algorithms suitable for linguistic applications derived from earlier work on UNION-FIND (Aho et al. 1976, after fellow Bell Labs researcher Robert Morris[44]). This method introduces the basic technique—which is present in most subsequent work—of processing the nodes of the input argument graphs in step, recursively. An equivalence class is built by combining arcs from each of the instant nodes. At each step, one of the input nodes is chosen as a representative of the class, and the other node is redirected, or *forwarded* to it. Where both inputs contribute an arc with a matching label, the procedure descends recursively on the representative of the joined inputs.

Most critically, immediately upon arriving at any node, its forwarding chain, if any, is followed to a terminal node, which is then considered a stand-in for the original node, including for the purpose of further forwarding. This is the fundamental insight of UNION-FIND, which I refer to as *dereference-before-forward*. With dereference-before-forward, membership in a co-reference equivalence class is signified by the identity of some terminal node. It follows that one-way (singly-linked) forwarding is sufficient to ensure that two disjoint chains—of any length, and which each represent distinct equivalence classes—can be joined by the trivial op-

[44] Aho credits Morris without citation. According to Doug McIlroy, also then at Bell Labs, Bob Morris' "path compression" idea was unpublished: "To Morris it was just one of those little tricks that one comes up with in the course of normal programming." (M. Doug McIlroy, personal correspondence 5/28/2011)

eration of setting the link field in one terminus to a value that identifies the other terminus.[45] Upon passing out of the input structures, the terminal nodes describe a graph which contains all of the arcs and reentrancies of both input graphs. Pseudo-code for the basic scheme, attributed to Boyer and Moore by Pereira (1985), is provided in Wroblewski (1987).

4.2.2 PATR-II: environments and structure sharing

The earliest linguistic uses of unification were strongly influenced by work in term unification. Definite-clause grammars (Pereira and Warren, 1980) and related term unification linguistic formalisms arose in response to advances in Prolog—such as Prolog II (Colmerauer 1982)—and the mature state of term unification research.

An early implementation in this vein was the PATR-II system (Shieber et al. 1984). This system uses for its term unification algorithm the basic procedure of UNION-FIND. To avoid the prohibitive expense of pre-copying every structure which participates in (destructive) unification, the system uses virtual-copy arrays, a familiar feature from Prolog, to virtualize the DAG representation. In another structure sharing scheme (Pereira 1985), TFS instances are ⟨skeleton,update⟩ tuples and the unification algorithm manipulates only the updates, which obtain context from their association with an environment, an input structure that is not altered. When accessing a derived instance, its updates must be applied, which may involve an $O(\log n)$ search at *each* node, and merging of environments.

As noted, Pereira's scheme is based on UNION-FIND, an algorithm that is necessarily destructive to at least one of the input graphs. This "ravaging" (Wroblewski 1987) of the argument graphs is not acceptable when they represent invariant reference structures—such as grammar rules or lexical entries—or memoized[46] parsing results. Since this describes virtually all of the unifications preformed in the normal course of grammar processing, destructive unification has limited utility in linguistic application.

The technique Pereira describes is complex, and certain of its aspects are justified, in part, by specific linguistic considerations from the parsing application. Perhaps the most important contribution of Pereira's work was to identify memory access *throughput* as the fundamental problem in unification. Although Pereira's response to the problem—that of crafting clever data structures that consolidate and defer these manipulations—was a worthy pursuit, he might not have suspected the severity of the issue: so fundamental is the throughput bottleneck in linguistic unification that most modern algorithms benefit from running each unification in two passes—the first to simply determine whether or not the costly operation might fail, in which case the second pass can be avoided altogether. It would be a few years before this type of approach was formally embraced.

---

[45] In fact, the forwarded node could be pointed to any member of the other node's chain.

[46] *Memoization* refers to storing intermediate results that may be needed in later stages of a computation.

4.2.3 D-PATR

In work developed across a few different implementation variations, Karttunen (1986) documents important early investigations of reversible unification. D-PATR was an influential unification parser which evolved from his earlier work at the Scandinavian Summer Workshop in Finland.

To prevent the destruction of grammar elements, one version of the system, Z-PATR, copies feature structures prior to invoking every (destructive) unification operation. An improvement in the D-PATR system hints at the insight—formally stated by Tomabechi a few years later—that unification failure is the operational case that should be favored in optimization. Specifically, D-PATR proceeds with the destructive operation forthwith, saving every modification it enacts to a list. In case of failure, these changes can be reversed, and an expensive, futile full copy of the input structures has been avoided. If the unification succeeds, it is the result structure that is fully copied, prior to reversing the changes which restore the input structures.

4.2.4 Incremental unification

Recall, with UNION-FIND, that the procedure was not usable because it ravaged one or both of the input graphs. An obvious workaround is to produce the result graph anew, without conscripting one of the argument graphs as a representative, a method explored by Wroblewski (1987). This method adds a COPY field to each node's scratch fields. This field allows a node in the result structure to be associated with the equivalence classes built by joining nodes from the input arguments.

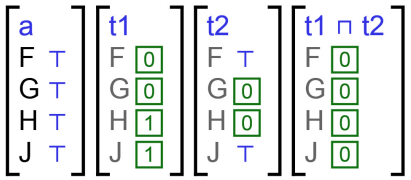


Figure 12. Coreference spreading. The unification of $t1$ and $t2$, each independently derived from a, equates two coreference equivalence classes which were distinct within $t1$. This transitive spreading can extend to arbitrary length.

Applying the efficiency categories he promulgated to his own work, Wroblewski concludes that, while his method eliminates early copying, it is still prone to over copying, specifically when joining coreference equivalence classes that were previously encountered and interpreted as distinct. Intuitively, we can see why any method that proactively creates result nodes will be subject to over copying: a greedy process cannot foresee whether those nodes may need to be joined later, meaning that only a single node should have been created. This is the problem of transitive coreference spreading (Figure 12), which directly inspired the *n*-way single-pass unification method detailed in Section 4.3.

Also widely cited (*inter alia*, Emele 1991, Tomabechi 1992, Callmeier 2000) from Wroblewski's work is the idea of the generation counter,[47] which became important to the bevy of methods that dedicate scratch fields—*in situ* within the input nodes—expressly to unification. In such schemes, it is important to be able to categorically abandon the values of these scratch fields, across all nodes in the system, so that subsequent operations do not consider them rele-

[47] Wroblewski attributes the idea to Ph.D. student Mark Tarlton, now a Distinguished Researcher at Motorola.

vant. Wroblewski notes that, by maintaining the value of a global sequence counter in each scratch slot—and only respecting the slot's contents if its value matches the current generation number—it becomes possible to globally invalidate all scratch slots by simply incrementing the counter.

The use of *in situ* scratch fields is not compatible with multi-threaded access to the structures involved. The easiest way to remedy this is to devise a method to associate a private set of detached scratch slots with each input structure. This is the technique used by van Lohuizen (2001) and by the array storage unifier described in Section 4.4. In detaching the scratch slots, the generation counter technique for categorical invalidation may become less important, as the slots can simply be discarded as a whole. Section 4.4.3 discusses such an approach in detail.

4.2.5 Lazy approaches

A number of different "lazy" approaches to unification have been investigated. In Godden's (1990) elegant variant, which the author contrasts with "eager" methods, the closure feature of the *LISP* programming language is used to capture the unification work required to join co-occurring input arguments. The hope is that the expense of the bulky language abstraction is outweighed by a reduction in unnecessary copying. In effect, however, the closure itself can be viewed as—already—a copy of the relevant result node. The effort undertaken to create the closure is wasted if the lazy result is never demanded, or if no further changes occur to the underlying nodes. Godden finds merit in his preliminary results, but the research avenue appears not to have been pursued further.

Tomuro and Lytinen (1997) investigate another method for deferring the wholesale copying of input graphs until a destructive modification is about to occur. The authors describe a copy environment which globally records a copy history for every top-level TFS that participates in a parsing operation. Each TFS's copy history contains a list of tuples which record the superseding of one of its internal nodes by some copy operation. These copy history entries are consulted when dereferencing nodes in various contexts.

Coming some years after the successful Tomabechi method, described in Section 4.2.9, Tomuro and Lytinen take aim at the assumption—inherent in Tomabechi's method—that failure is fundamentally more frequent than success in linguistic unification. It is true that any two-pass method strongly implies the incorporation of this tenet, but the authors of the newer work do not mention whether they have encountered a realistic natural language grammar which challenges it.

More significantly for the method proposed in this thesis, Tomuro and Lytinen also question the "extra bookkeeping" required by Tomabechi's method, suggesting that his approach is necessarily complicated by its two-pass character. Although further elaboration on this point is not provided, it is safe to assume that the authors are referring to the COMP-ARC-LIST in Tomabechi's method, a data structure which propagates result arcs—representing features that are

not covered by the scratch slots of the designated representative node—from the first pass to the second pass. A key contribution of this thesis is the result that this complication can be eliminated from a two-pass method. The array TFS unifier, presented in this thesis, demonstrates the technique. Although still subject to Tomuro and Lyntinen's first objection, the present work—by describing a two-pass method that is simpler than any published lazy method—disqualifies their second.

4.2.6 Chronological dereferencing

Noting that Wroblewski's generation counter describes a sort of chronology of TFS changes, Emele (1991) describes a method which seeks to mitigate the third type of categorical inefficiency—redundant copying—by associating a snapshot of the generation counter with a "copynode." Copynode environments are extended—and the generation counter incremented—whenever a destructive change is about to occur. In other respects, the copynode is similar to the environment used in earlier methods (*viz.* Pereira 1985). Node dereferencing is extended so that it not only traverses across nodes, but across the generation chronology.

The system is designed for the storage of structures that persist outside the context of unification, but as a bonus, the same mechanism facilitates unification itself. A potential disadvantage of the system is the unchecked accretion of copynode layers that might occur when building elaborate structure, such as during parsing. As with the earlier environment methods, following a lengthy chronological chain exacts penalties during node dereferencing.

4.2.7 Later work in term unification

In term unification—a special case of the graph unification discussed in this thesis—the list of features appearing in each unification argument must be identical in order for the unification to succeed. Because they work with isomorphic binding lists, term unification systems are simpler and thus are likely to exhibit a performance advantage over graph unifiers. Carroll (1993) compares his ANLT term unification and parsing system with the contemporary work, noting that term unification parsers are little studied (*ibid*, 40). This remains the case today; modern-day linguistic research has largely adopted graph unification methods. To a limited extent, one advantage of term unification cited by Carroll—that every category contains only a fixed set of features—can be enjoyed by graph unification implementations which exploit the grammar's feature appropriateness condition. The method contributed by this thesis is a concrete example of such an approach.

4.2.8 Strategic lazy incremental copy graph unification

The 1990s saw continued advances in linguistic unification research. With his LING system, Kogure (1990) describes a technique which enables structure sharing while retaining $O(1)$ node access performance, a substantial improvement on earlier structure sharing methods (*viz.* Pereira 1985). Structure sharing addresses the problem of redundant copying—the waste of time and space associated with reproducing any structure which appears elsewhere in the grammar. From a baseline of Wroblewski's method, LING adds a field to each node that tracks

dependencies between structures. A special routine for copying nodes examines these dependencies to determine the cascading effects of a modification. This permits maximal structure sharing without worrying about spurious sharing, but the complexity of copying increases.

The same report also discusses the SING unifier, which incorporates a stochastic model to predict likely unification failure paths, so that these paths can be followed first. Kogure's two methods are combined in the strategic lazy incremental copy graph (SLING) system, but no evaluation is provided.

Methods similar to SING are commonplace in today's systems. Quick-check (Kiefer et al., 1999, Malouf et al. 2000) is implemented in all DELPH-IN parsers, including *agree*, the new system presented in this thesis. Like Kogure's system, quick-check uses offline training to develop a set of failure-prone feature paths.

In a similar vein, *agree* also offers an adaptive (unsupervised) tuning feature which operates on single-feature frequencies. When enabled, an independent tally for the immediate feature which causes each unification failure is incremented. After each sentence (in batch parsing), or at some other prescribed interval, the persistent feature evaluation order used by the unifier is adjusted according to descending order of these tallies. The simple method shows promise but is not discussed further in this thesis; evaluation is left for future work.

4.2.9 Quasi-destructive unification

Contemporary with Kogure's work, Tomabechi developed "quasi-destructive unification" (Tomabechi 1991, 1992), a method which became one of the most cited. This is the method implemented in the LKB (Copestake 2002b) and PET (Callmeier 2000) parsers. The key insight of this work was to explicitly acknowledge that unification failure is far more commonplace than unification success in realistic linguistic application. To avoid over copying, therefore, unification should occur in two passes, with no memory allocations occurring during the first pass.

Like the array storage unification presented in this thesis, quasi-destructive unification is based on UNION-FIND (Aho et al., after Morris 1976) and comprises two synchronous passes. The first pass joins the input structures by manipulating *in situ* scratch fields, but never writes any output structure. The argument TFSes are recursively traversed from their root nodes, in tandem. At each step, nodes are paired according to their co-occurrence in the traversal. When arriving at a pair, each node's forwarding pointer is individually dereferenced, if non-terminal, by following a forwarding chain until reaching a terminating node. The terminus of the forwarding chain, and its corresponding scratch slot, replace the original node for further operations with the node pair. Next, pre-existing node identity and type unification (or failure) are checked between the dereferenced pair. If type unification succeeds and the nodes are not already joined, then one of the nodes is designated as the representative, and the forwarding field in the other node's scratch slot is pointed to the representative.

In order to avoid modifying the input TFSes, the first pass records all of its modifications in scratch fields; as in Wroblewski's method, a global generation counter allows them to be summarily discarded should unification later fail. The scratch fields consist of the node forwarding pointer (from UNION-FIND) and a list called COMP-ARC-LIST. Between the feature-value pairs of the instant nodes, the method calculates the set intersection and the set complement. The former set is maintained in the representative node, with modifications handled according to the forwarding pointer. The pairs from the latter—that is, features that are only constrained by the input argument that was not chosen as representative—are added to the COMP-ARC-LIST, which is then considered adjunct to the representative. The joint traversal continues recursively on the set of feature-value pairs constrained by both nodes. TFS realization occurs in the second pass of the algorithm, which is only undertaken after overall success has been guaranteed.

Tomabechi emphasizes that no output structure is written in the case of unification failure—thus, there is no "over copying." This should be optimal assuming his observation about the preponderance of unification failures. For the decade that followed, his observation inadvertently galvanized much of the subsequent linguistic unification research around the notion of avoiding unification *altogether*, rather than optimizing its enactment. The culmination of the trend was the description of several pre-unification checks (Kiefer et al., 1999, Malouf et al. 2000), notably a rule compatibility pre-filter and the trivial—but highly effective—quick-check technique. In this latter technique, an empirically-tuned set of likely failure paths are probed for failure of type unification, prior to initiating unification.

Pre-filtering techniques—and unification research—quickly and properly became preliminary and adjunct to unification proper, with the result that Tomabechi's method has remained the staple unifier for many modern grammar engineering environments, including all prominent DELPH-IN parsers: LKB, PET, and Ace.[48] What has not been re-examined in the linguistic unification literature is where the quasi-destructive unifier—and two-pass parsing in general—stands after the widespread adoption of effective preprocessing filters, a question left for future evaluation.

I conclude this section with mention of a pertinent issue which relates to the present thesis. In the most general sense, Tomabechi's method describes two separate "forwarding" mechanisms. The scalar forwarding pointer, inherited from UNION-FIND, forwards *nodes*, while COMP-ARC-LIST "forwards" *arcs*. This dichotomy is examined in Section 4.4.1, which culminates in the description of new unification algorithm, adapted to array TFS storage, which produces a simpler, more consolidated representation of the result TFS. The array TFS unifier exploits a simplifying guarantee offered by the storage model, resulting in the assurance that operationally variant auxiliary data structures such as the COMP-ARC-LIST described in this section are not

[48] Ace is a DELPH-IN compatible parser and tactical realization system developed by Woodley Packard. http://moin.delph-in.net/AceTop, http://sweaglesw.org/linguistics/ace/

needed. The next section concludes the chronological review of the TFS unification literature with a discussion of the application of computational concurrency to the problem.

4.2.10 Concurrent and parallel unification

After a decade and a half of gains in unifier efficiency, research began on unification algorithms that permit multiple concurrent accesses to communal input TFSes. In the terminology of van Lohuizen (2000), *parallel unification* refers to the ability to schedule multiple processors on the same unification task, whereas *concurrent unification* refers to the ability to schedule multiple single-threaded unification tasks—involving the same grammar instance and/or parse chart—on multiple processors. Thus parallel unification is more fine-grained than concurrent unification. van Lohuizen provides an exhaustive analysis of the inter-CPU communication implications of finer-grained tasking.

Any method that treats publicly visible data—such as canonical TFS instances—as read-only is inherently thread-safe. Unfortunately, this criterion is not met by many of the unification algorithms reviewed above, including Tomabechi's quasi-destructive method, which is widely used. Methods which predicate their efficiency upon exclusive access to express scratch fields in the input TFSes are, by that same requirement, rendered unsuitable for concurrent use.

A natural solution to this problem is to detach the scratch fields from the nodes of the input structures (van Lohuizen 2000). This requires adopting an efficient and perfect index which associates each node of a given TFS with a distinct scratch structure. *Efficient* means that finding the scratch slot for a node should be fast, and that few of the allocated slots should remain unused. *Perfect* means that the mapping should be distinct (no slot is assigned to more than one node) and unique (no node is given more than one slot).

This reduces to the general TFS feature arity problem (Section 2.2.3), to which array TFS storage—presented in section 2.3—describes a solution. Over this storage, *agree* implements a new, thread-safe unifier. The *agree* unifier achieves this via the simple means described above: by detaching the unifier's scratch slots so that input structures are not modified in any way. The precise method of mapping, presented in detail in Section 4.4.2, is contrasted with van Lohuizen's approaches next.

With any type of TFS storage, a simple method of detaching scratch slots is to assign a unique index from an integer sequence to each node of a TFS. This requires a traversal of the structure, an operation of no consequence for persistent grammar structures, but which accumulates a burden for structures built during parsing. Using such an index is trivial; the TFS is considered read-only, and any threads that wish to unify with the TFS obtain an array of private scratch slots which remain isolated for the duration of their operation.

van Lohuizen investigates two improvements on the basic node numbering idea, and evaluates them in his CaLi concurrent unification parser. One scheme simply hashes the node pointer,

and this likens it to the array storage method proposed in this thesis, whereby scratch slot mappings are co-opted wholesale from the underlying array storage, which, in turn, hashes a node-feature tuple $\langle q', f_i \rangle$ to obtain the (array storage index of) a node, q. By piggybacking on the existing array storage mapping, my scheme offers the advantage that the scratch slot index is a free by-product of the node lookup that has to be performed anyway.

For the second method, van Lohuizen describes a scheme of numbering nodes with relative offsets, such that nodes of different graphs obtain unique scratch buffers, with no collisions. As a happy side-effect, both schemes preclude spurious sharing, allowing the restrictions on structure sharing discussed in Section 2.2.2 to be relaxed (van Lohuizen 2001, 78).

Returning to the discussion of van Lohuizen's distinction between inter-unification concurrency and its finer-grained sibling, intra-unification parallelism, I concur with van Lohuizen when he notes that a detailed research agenda for the latter seems premature:

> as long as the number of unification operations in one parse is large, it is preferable to choose concurrent unification. Especially when a large number of unifications terminate quickly…, the overhead incurred by [parallelism] can be considerable. (van Lohuizen 2001, 72)

This concludes the review of prior work in TFS unification algorithms. The next section describes an unusual approach to TFS unification which was investigated during the research for this thesis. In particular, by describing the optimal unifier for a certain theoretical class, the work establishes an upper bound on the performance of single pass unification.

4.3 *n*-way unification

The emphasis in the unification literature on categorizing unification algorithms according to classes of copying behavior suggests an approach to unification that trivially minimizes *over copying*—the production of any structure which does not contribute to the result graph—by producing result structure only when it is provably definitive. For example, recalling the "lazy unification" work of Godden (1990), consider a lazy method—that aims to skip effort that might late be revised—but with perfect foresight about whether or not such revision will actually occur. With perfect knowledge of whether any given piece of substructure is be definitive (or not), an algorithm can ensure that every task that the unifier undertakes is productive, moving forward to either unification success—or the determination of failure. To implement this idea, a new unification algorithm—*n*-way unification—was developed and evaluated. The following sections describe this method.

Although—for engineering reasons discussed at the end of this section—the method was eventually abandoned in favor of the two-pass array storage unifier described in Section 4.4, it performed nearly as well. Results from an evaluation of the work will also be summarized. These are supportive of the hypothesis, suggested at the end of section 4.2.9, that the empirical con-

siderations that originally motivated Tomabechi's emphasis on two-pass unification may have since been eroded by effective unification pre-filtering.

4.3.1 Motivation

Despite being deemphasized in *agree*, *n*-way unification remains significant as an expression of two theoretical bounds on single-pass unifier performance. First, depending on interpretation—discussed below—the method exhibits neither early copying nor over copying. Second, the algorithm traverses the theoretically minimal number of links (that is, issues the minimum number of recursive calls) required for a unification operation. Specifically, worst-case node visitation is $O(n)$ in the number of input arcs, regardless of the number of reentrant nodes. It achieves this by consolidating the complete, arbitrarily-sized set of reentrancies into a single high arity function call. UNION-FIND based methods tend towards $O(n^2)$, because dereference-before-forward can involve following a chain through $n - 1$ nodes (in the worst case) for each node. This analysis matches the intuition that, in a duplex (two-way) unifier, each recursive function call can only join, at most, a single node to an equivalence class, and equivalence classes can be arbitrarily extensive.

Because it is a single-pass algorithm—the writing pass is also the visitation pass—*n*-way complexity is also favorable for writing the result structure, never traversing more than the $O(n)$ edges of the *result* graph.

The key idea behind *n*-way unification is that it is preferable to simultaneously unify the set of nodes that form a *coreference equivalence class*. To do this requires definitive knowledge of whether each equivalence class has coreferences outstanding, so as to defer descending into it. The algorithm tracks whether an equivalence class is complete (meaning that it cannot possibly obtain additional members); if this is not the case the class is summarily skipped. Only when complete does it undertake the class's unification, emit a definitive result node, and descend into its substructure. Each descent takes the form of a single variadic (variable-arity) function call. For each deferred class, two bookkeeping items are maintained: a list of nodes comprising its membership thus far, and a simple integer tally representing the number of reentrancies remaining until downwards descent is permitted. Ideally, the list of nodes directly implements the format of a variadic stack frame for the (eventual) recursive call.

4.3.2 Procedure

n-way unification can be outlined very simply. The algorithm traverses the argument TFSes in step, accumulating the *n* coincident argument nodes and summing their expected reentrancy counts (minus one, for each, to reflect the current visit), or—for previously seen nodes—decrementing the existing sum. If this total is zero after all of the input arguments have been processed, then the class is definitive: no further nodes can possibly join the equivalence class,

and the node type will be the greatest lower bound between the class members' node types.[49] Only in this case is the variadic gang-descent below the nodes undertaken. The accumulated set of nodes is simply submitted as the variadic argument set of a recursive call. In a preferred implementation, the set has been maintained in a form immediately suitable for this. Termination is signaled in the same way as any depth-first traversal: by completing the processing of all of the root node's feature-value pairs.

Since writing a node is definitive, as soon as this determination is made, its scratch data is no longer needed, and the algorithm might as well write the result node, if this can be done cheaply and in a manner that permits trivial discarding. As noted in Section 4.3, doing so raises interesting questions concerning the performance categorization model traditionally used in unification research.

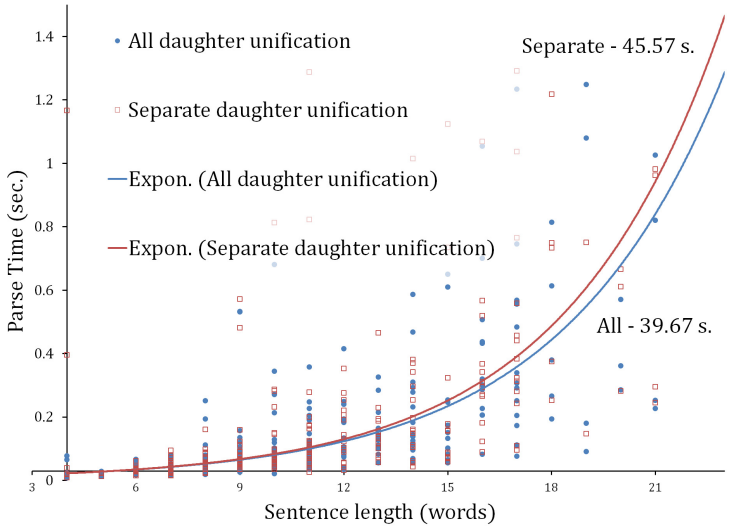


Figure 13. Unpacking comparison of simultaneous-daughter versus separate-daughter unification. The *n*-way unifier is used to exhaustively validate all of the derivations a packed parse forest. In the red (upper) line, it simulates duplex unification, so multiple top-level unification calls are required to validate binary rules. The blue (lower) line shows that validating the entire corpus is faster when the unifier is permitted to unify binary rules all at once.

The tested implementation of *n*-way unification does not, in fact, write array storage records in their final, persistent storage location, opting instead to image array TFS nodes—in *nearly* final form, but without their hashing data—onto an upper stack frame of the unifier thread. By this technique, *n*-way unification avoids a heavy-weight system array allocation in the case of failure. If unification succeeds, the entire block of nearly-complete result nodes is copied to persistent storage with a single `memcpy` instruction, and the hash index is built. Because it does not perform system allocations, this seems to qualify as 'no early copying' under the spirit of that definition, but clearly the `memcpy` and hashing would then count as a second pass, albeit a speedy one.[50]

n-way unification also provides an elegant means for introducing additional TFS arguments during unification. This ability accommodates the formalism's well-formedness condition

[49] If sufficiently many diverse types are present at this step, a single multi-type GLB calculation via bit-array operations (Aït -Kaci 1989) may outperform multiple pairwise retrievals from a (duplex) GLB cache.

[50] This is a simple $O(n)$ loop over the array, rather than a graph traversal.

(Section 2.1.5), and, by permitting the simultaneous unification of all of a rule's daughters,[51] also improves the efficiency of validating derivations in a packed parse forest. The improvement is shown in Figure 13. Simultaneously unifying all of a rule's daughters with a single unifier operation improves total parse time for the 'hike' corpus by 13%

The main cost of *n*-way unification is that each TFS that participates in unification must provide to the algorithm an accurate set of tallies corresponding to the number of reentrancies expected for each coreference.[52] Although there are administrative costs to maintaining these structures, each invariant set is trivially produced during the (*n*-way) unification that creates the structure it summarizes, and is thereafter valid for that structure's lifetime. For the prolifically referenced structures that constitute the grammar itself, the amortized cost of computing these tallies is zero.

The method summarily skips incomplete classes, and this leads to a surprising and useful property: with monotonic traversal alone—no backtracking—there is an adequate completeness guarantee. Walking forward through the input structures, any skipped node will always be found accessible via some outstanding reentrancy—in the *remaining* unvisited structure—of *some* argument TFS. This is the condition which limits the worst-case traversal to $O(n)$ in the total number of input arcs. Input structures with cyclic (or more relevantly—'mutually-cyclic'—since the DELPH-IN formalism does not permit cycles in any completed, well-formed structure) coreferences violate the condition, but these cases are easily detected: if, upon reaching the end of the traversal, any classes have non-zero reentrancy counts, then this can be taken as a true positive indicator of unification failure owing to cyclic structure, with the peculiarity that the exact failure path is not necessarily known.

The case described above—failing to visit substructures that express mutually cyclic coreferences—occurs quite infrequently, even in large grammars such as the English Resource Grammar. But the case does highlight another illuminating characteristic of the *n*-way method: on the one hand, unification unnecessarily proceeded to completion, when perhaps it could have been abandoned earlier had the cycle been detected. But on the other hand, a potentially large chunk of substructure was skipped in reaching this 'completion.' Whether this obscure trait is an advantage or a disadvantage, therefore, will be grammar-dependent.

4.3.3 Evaluation

Evaluation of *n*-way unification was undertaken by comparing it to an earlier unification implementation which was available in the *agree* grammar engineering system. In fact, the array

[51] While the improvement is applicable to unpacking, it is not generally useful in parsing, where the daughters of binary (or higher arity) rules must be unified individually so that partial structures ("active" edges in the parse chart) that support the pursuit of alternate parse hypotheses can be retained.

[52] This is the number of immediate arcs leading to a coreferenced node, not the number of TFS paths leading to the node. The latter can be greater than or equal to the former.

storage method and its accompanying unifier represent the fourth iteration[53] in the design and development of *agree's* unification. Figure 14 compares the *n*-way unifier with the second unifier in this sequence, an implementation of Wroblewski's incremental method.

4.3.4 Classifying unifier performance

The nature of *n*-way unification confuses the discussion of early copying and over copying. Recall the comparison with the "lazy" method mentioned in Section 4.3, the introduction to *n*-way unification. When every step in unification algorithm is provably useful, it seems foolish not to produce the result structure along the way (if it can be done very cheaply) even if failure is eventually determined. Because the algorithm never returns to a node once it is written, writing the result could be considered a substitute for the manipulation of the scratch fields of other methods, an activity which is apparently exempt from categorical review.

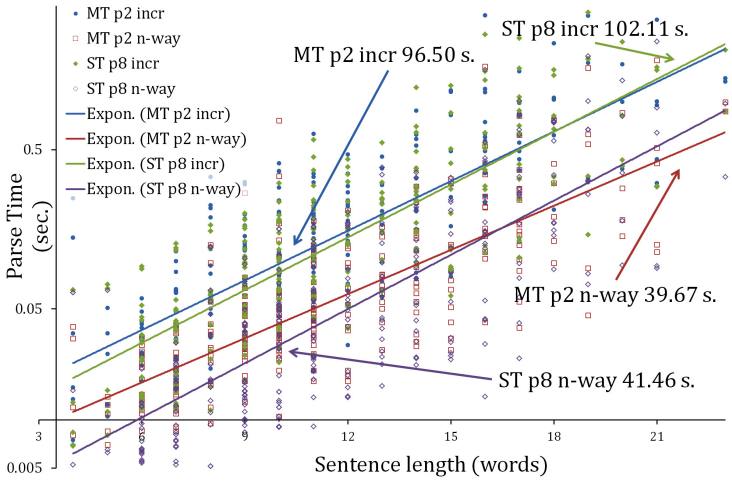


Figure 14. Intra-*agree* evaluation of *n*-way unification *vs.* incremental unification (Wroblewski 1987) for parsing and exhaustively unpacking a subset of the 'hike' corpus. Multi-threaded, dual pipeline (MT p2) *vs.* single-threaded octal pipeline (ST p8). Methodology details are given in Chapter 5. *n*-way outperforms the incremental method, especially when multi-threading with long input sentences. For comparison purposes, note that the array storage unifier presented in this thesis completes this particular task in 35.4 seconds (MT p1).

It is worth examining why such an approach should appear to challenge *de facto* unification research dogma, which holds that no result structure should be produced if unification does eventually fail. The chief question raised here is, precisely what distinguishes "result structure" in the definition of *over-copying*, and why should its production be treated differently than the manipulation of temporary "scratch" fields by the categorization model?

At issue is the nature of the control data which every unification algorithm carries to facilitate its operation. The same measurement bias which ignores the manipulation of scratch fields as a performance hazard category allows the *n*-way method to wholly subvert the informal performance model. By taking the injunction against over-copying to an extreme, the *n*-way technique demonstrates a weakness in the model. Methods with substantial and complex control

[53] This does not include a fifth implementation, a variation of the *n*-way code which proceeds only downwards on the thread's call stack, so that when passing out of the traversal, the result TFS is entirely encoded on the deep set of calling stack frames. The method, which completely eliminates the need to pre-allocate scratch structures, has the advantage that the size required for the TFS becomes known at that deepest stack frame. The persistent storage for the actual final TFS is allocated at that time and the TFS is realized at each step of returning from the nesting. After completing a working prototype, I abandoned the method because evaluation suggested the overhead of traversing $O(n)$ stack frames (in the number of result graph nodes) would be limiting.

structures, like *n*-way and Godden's (1990) lazy method subvert the model by deploying analytical techniques which make *too much effort* to reduce high-profile performance hazards, and thus succeed only in shifting complexity to obscure hiding places. Accordingly, the field may benefit from the development of a more rigorous unifier performance evaluation model.

4.3.5 Summary

To conclude the discussion of *n*-way unification, I mention some engineering details that significantly hampered the implementation. The preceding discussion does not describe the manner in which the variadic function call is implemented, and this is critical to the performance of the method. Although the algorithm is theoretically elegant, efficient implementation in the CLI runtime environment proved a challenge. That the *n*-way method nearly matches the performance of the two-pass array storage unifier is surprising, because the source code for the former is so much more belabored and overwrought. Either algorithm could have been carried forward in *agree*, but—even discounting the need to maintain per-TFS coreference tallies, which I generally did not consider a demerit against *n*-way unification—in the end, the *n*-way implementation just required too many unappealing platform-induced workarounds.[54]

Future work will investigate whether the theoretical elegance of *n*-way unification survives better under alternative tooling. In any case, the result supports the hypothesis, mentioned at the beginning of this section and detailed in Section 4.2.9, that unification pre-filters have eroded the original motivation for two-pass unification to such a degree that the overhead of its extra traversal has become a more significant liability, allowing efficient single-pass methods, such as *n*-way, to become relevant again. In effect, pre-filtering increasingly serves the function of the first pass and diminishes the effectiveness of strategies that incorporate a failure-based elimination pass.

The next section presents the new unification algorithm adapted for the array TFS storage presented in Section 2.3. This unifier supports concurrent operation and improves on Tomabechi's method by eliminating the need for post-initialization operationally-variant list structures such as the COMP-ARC-LIST.

4.4 Array TFS unification

This section describes a unifier adapted to array TFS storage. Like Tomabechi's (1991, 1992) method, the technique is based on UNION-FIND, the 1976 technique introduced in Section 4.2.1 and described in Section 4.2.9. The array storage unifier differs from Tomabechi's method in several ways. A key contribution of the new method is a novel method for indexing detached scratch slots, so as to permit concurrent unification.

As noted in the Section 4.2.10, while the algorithm is universally thread-safe, it is also passive, meaning that it has no provision for dispatching additional sub-tasks to assist with a portion of

[54] A charitable description of the ugly hacks and kludges involved.

the unification job with which it was initially charged; nor does it capture or retain threads for any purpose. It permits concurrency only by virtue of its thread-safety; and becomes concurrent only as so-utilized by the caller. In *agree*, the caller is most often the concurrent chart parser, which in fact heavily exploits the unifier's thread safety by deploying numerous simultaneous unifications.

A second contribution of the new method—related to the scratch slot assignment technique—is a method for the elimination of excess bookkeeping structures. Section 4.2.9 observed that, in Tomabechi's method, the forwarding pointer mechanism and the COMP-ARC-LIST mechanism seem, to a certain degree, to overlap. This section completes the earlier analysis, presenting a non-destructive, two-pass unification algorithm that exhibits no over-copying and does not maintain a list of extraneous arcs for each result node.

Notably, a powerful guarantee emerges when invariant arity mappings are co-opted from the array storage system for use as pre-made structural descriptions of each input argument. Ultimately, the guarantee allows the unifier to adopt a simplified conception of its task as one of selecting from the small number of options available at each node—as opposed to the more conventional idea of a unifier as an algorithm that must be able to "build" new structure. Freed of excess data structures which anticipate the synthesis of arbitrary new structure, the complete and essential work product of the unifier is reduced to a single set of scalar forwarding pointers relative to its array of scratch fields. In short, the guarantee is that the result structure is necessarily latent within the joint set of array storage layouts of the TFS arguments, from the very instant they are co-opted, and so there is always a sufficient read-only substrate available, ready-made for efficiently expressing the result structure prior to the start of the operation. Naturally, this implies that no manner of operationally variant allocation is needed, and the progress of the operation itself is generally streamlined.

4.4.1 COMP-ARC-LIST

In Tomabechi's quasi-destructive unification, the function of the COMP-ARC-LIST is to gather, for each node in a TFS, zero or more feature-value tuples which are required in the output node, but which are not present in the designated representative node. The need arises in two situations. Consider the unification of nodes q and r. In the first situation, neither q nor r constrains a set of features which is a superset of the other:

$$\left|\pi_f(q.A) \cup \pi_f(r.A)\right| > \max(|q.A|, |r.A|). \tag{4.1}$$

In a second situation, the type unification result between two nodes is a type which is identical to neither of the argument nodes' types:

$$q.t \sqcap r.t = g, g \neq q.t, g \neq r.t. \tag{4.2}$$

When this situation occurs the well-formedness requirement is triggered, meaning that the canonical constraint for type g must also be unified with $q \sqcap r$. In either situation, neither q nor r contains a complete set of slots sufficient for holding the features needed in the result. Alt-

hough the Tomabechi method wisely seeks to avoid allocations during the compatibility checking pass, it does need to somewhere maintain a consolidated set of features for each developing result node. Since only one of the nodes can be designated, its list of arcs is supplemented with those from the COMP-ARC-LIST.

My analysis begins with equation (2.19) in Section 2.2.3, repeated here, which suggested an informal conception of TFS storage where features alternate with nodes:

$$\big((q\ f),(q\ f),(q\ f)\ldots\big) \tag{4.3}$$

In Tomabechi's method, each node q contains a forwarding pointer, deployed as an *in situ* scratch field, which allows the unifier to forward any node to another. Meanwhile the COMP-ARC-LIST mechanism handles the features (or 'arcs') needed—but not available in—the target of the node forwarding.

Consider now if unifier scratch slots, rather than existing within each node, were instead available according to the groupings as parenthesized in (4.3). This shift entails that there would be a unifier scratch slot corresponding to every feature arc, as opposed to every node. This, in turn, means that it is a *feature arc*, rather than node containing it, that contains the scratch fields for the node it *leads to*—and that a separate mechanism for temporarily storing forwarded arcs, the COMP-ARC-LIST of the prevailing method, would be unnecessary.

An immediate problem with this idea is that a coreferenced node has several feature arcs leading to it, and it is not clear which of these should serve as its unique scratch slot. But this issue is reminiscent of how array TFS storage, as detailed in section 2.3.4, replicates, *by-value*, a given coreferenced *out*-tuple for each feature arc—or *in*-tuple—that references it. The design, whereby graph reentrancy is modeled by the (value, as opposed to referential) equality of *out*-tuples across relation $\mathbb{A}$, and where conflation of coreference thus becomes a task external to the storage paradigm, actually matches the shifted scratch slot condition suggested here for simplifying Tomabechi's algorithm.

It should now be clear that the key insight is for the unifier to exploit the pre-existing storage layout of any given array storage TFS—to be precise, its *in*-tuple hash table—for the purpose of arranging a unique scratch slot for each feature arc in that TFS. In doing so, the new unifier assumes responsibility for ensuring the conflation of joined *out*-tuples, but because coreferenced nodes are marked by a flag bit in the type identifier, this is trivially accomplished.

Each slot mapping co-opted from an array storage TFS represents a fixed grouping of substructure slot mappings which reflects the actual feature expression of some node in one of the input argument TFS. When superimposed on the global set of scratch slots which incorporate multiple argument TFS layouts, these slot mappings can be intricately interconnected to assemble a representation of the result graph using only a single forwarding mechanism.

Because the co-opted mappings represent pre-configured, content-based retrieval keys for each input argument, another way to look at the idea is as a shift in the nature of the node retrieval key, so that it is *arcs* that are forwarded, rather than *nodes*. Because arcs (*in*-tuples) explicitly incorporate a binding to their target node, they also carry an implicit association with that (target) node's other inbound arcs (if any—that is, if it is coreferenced).

By carefully exploiting the existence of these pre-configured, implicit groupings, a single forwarding mechanism can describe the result configuration. Informally, instead of a conception where a list of zero or more $\langle f_i, q_i \rangle$ tuples is retrieved from each node q', the simplified scheme imagines that only a single node q_i (or no node) is retrieved from each $\langle q', f_i \rangle$ tuple. The next section continues with a formal treatment of the method.

4.4.2 Scratch field mapping

van Lohuizen importantly noted that the fundamental problem in adapting a Tomabechi-style algorithm for parallel operation is to associate each TFS with a set of scratch fields that are private to each concurrent unification operation. More specifically, within this private set, each TFS node must be uniquely mapped to a distinct scratch slot. *Uniqueness* ensures that coreferenced nodes are correctly conflated, and *distinctness* ensures that non-coreferenced nodes are not incorrectly conflated.

It is obvious that, with array storage, a set of indexes is already implicit in the underlying storage organization. By adding the ability to query the storage index of a 4-tuple, these can form the basis for identifying a detached scratch slot with each node. To facilitate the discussion, the relational algebra expression for an array storage 4-tuple is extended to incorporate its array storage index, ix,

$$a_i = \langle {}_{(\mathbb{Z})}ix, {}_{(Feat,\mathbb{Z})}\langle f, m_F \rangle, {}_{(Type,\mathbb{Z})}\langle t, m_T \rangle \rangle \tag{4.4}$$

which has flattened form $a_i = \langle\, ix, f, m_F, t, m_T \,\rangle$. As before, this notational extension does not imply a change in the underlying storage. In this case, rather, the notation is extended by making explicit an intrinsic property of the storage.

Although the set of node-index mappings

$$\underset{ix,\, m_T}{\pi}\, \mathbb{A} \tag{4.5}$$

satisfies the distinctness requirement, it does not ensure that each node has a unique index. Recall that, in array storage, the single 4-tuple which corresponds most canonically to a particular node q is *not* contained in the set of 4-tuples selected by its *out*-mark:

$$\underset{m_F = q.m}{\sigma}\, \mathbb{A} \tag{4.6}$$

This expression selects a number of array storage rows which represent the feature-value pairs, or onward 'arcs,' for q, but the storage location for the node itself, that is, a row which stores

its type, cannot be recovered without further maintaining context. To be precise, we seek one of the 4-tuples selected by a governing node q' and one of its appropriate features, f':

$$\underset{f = f' \wedge m_F = q'.m \wedge m_T = q.m}{\sigma} \mathbb{A} \tag{4.7}$$

Although each row selected by this expression contains the same *out*-tuple value—specifically, $\langle q.t, q.m \rangle$, or simply $\langle q.t, m \rangle$—there may be multiple such rows, each representing different *in*-tuples belonging to different nodes $q'_0 \ldots q'_n$. These represent the one or more feature paths from $q_i{}'$ to q. This is no problem for ordinary TFS access, where value identity amongst the *out*-tuples is sufficient to guarantee correct behavior—after all, the copies are indistinguishable. In fact, the fundamental design of array TFS storage scheme is optimized to serve precisely that case: by having an identical copy of the *out*-tuple installed at each feature path, an additional join or array fetch is avoided, per access. But when these nodes' array storage indexes—rather than their values—are examined, the illusion of identity, or *referential equality*, is lost, violating the uniqueness condition required of a scratch slot mapping.

Summarizing this, the underlying reason that a node's array storage index is not ready-made as a scratch slot index is that the set of *out*-tuples is not a true relation. In order to avoid an extra join operation and an extra object allocation, array storage fuses the *in*- and *out*-tuples into permanently bound 4-tuples, with the result that coreferenced *out*-tuples are explicitly duplicated. To clients of the array storage system who receive and process *out*-tuples (and the mark values they contain) this fact is invisible, because numerical mark values naturally conflate. But consumers of the array storage *index* experience that the *index values* for nodes that are supposed to be coreferenced are *not* conflated; each is reported as having a distinct index depending on the governing 4-tuple from which it was reached. If the unifier wishes to co-opt array TFS storage indexes for the purpose of scratch slot mapping, it concomitantly assumes the responsibility for the conflation of coreferenced nodes.

One way to effect this conflation would be to privilege one of the rows by fiat. For example, one could specify that the lowest index value

$$\min \sigma_{a.m_T = m} \mathbb{A}\, a.ix \tag{4.8}$$

serve as the scratch slot index for the set of all rows which contain $\langle q.t, m \rangle$. With such an edict, scratch slot uniqueness is restored, and unification will correctly experience the conflation of coreferenced nodes. This particular solution, however, is not workable. When working with node $q = \langle t, m, A \rangle$, the relevant storage index is the index from which the governing *out*-tuple $\langle q'.t, q'.m_T \rangle$ was retrieved. This is the origin of t and m, passed down from a higher caller, and incorporated as $q = \langle t, m, \ldots \rangle$.[55] Certainly $q'.ix$ could be passed down from the caller along with t and m. But a more serious problem foils the fiat plan: there is no practical way for the callee—or the caller, in fact—to determine whether $q'.ix$ has the lowest value amongst its set

[55] Recall that the storage indexes associated with rows $q.A$, which are easily accessible, are unrelated to the notion of a canonical array location for q. For one thing, there may be no such rows.

of matching *out*-tuples, because this set is not available. The set would be expensive to compute, and is of no use to general-purpose TFS operations.

A second solution to the index uniqueness problem—which reflects the implementation of the system evaluated in Chapter 5—takes advantage of the convention that coreferenced *out*-tuples are always given a mark values less than zero. As is required in order to co-identify nodes which are coreferenced, a unique negative mark value is assigned even when the type of the node has no appropriate features, which would normally require that the mark value be zero. The details of this were discussed in Sections 3.2 and 3.2.3. By giving special treatment to rows where the mark in the *out*-tuple is negative, clients who use storage indexes for their own purposes—such as the unifier described here—are able to manually effect the necessary conflation of nodes. Further details are given in Section 4.4.4, which describes scratch slot manipulation.

The latest implementation of *agree* implements the most elegant solution to the problem.[56] In this method, also mentioned at the beginning of Section 4.5, the unifier simply uses its own forwarding mechanism to restore the referential equivalence, within its scratch slots, of the array storage *out*-tuples. Doing so requires an array of pointers per TFS, per coreference, that store the identity of the first slot that is encountered from each equivalence class and which serve to map the negative *out*-mark (treated here as an array index) into the forwarding chain during the subsequent encounters. Because each array TFS carries, as an invariant property, the number of coreferences it contains, this adjunct list is trivially established prior to commencing the unification, and is not considered an operational variant which would violate the tenet that the array storage unifier does not synthesize. The result of this scheme is that the scratch slot assignment for each unification argument has two sections, not three. A section dedicated to coreferenced nodes is no longer needed, and only a single slot (for the root node) and the main range which parallels $\mathbb{A}$, are needed. It is clear by now that unifier scratch slots are key to most unification algorithms. The next section discusses scratch slot allocation and the discarding problem.

4.4.3 Scratch slot initialization and discarding

A major difference between array TFS storage and van Lohuizen's approach to scratch slots is that van Lohuizen maintains a limited set of scratch buffers in a single, global pool. While it is crucial that frequent allocations be avoided, this solution introduces contention for the leased slots in the common pool amongst concurrent unification threads. Also facing contention is the global generation counter value which controls the invalidation of every slot in the pool. [57]

---

[56] The technique described in this paragraph is not included in the system evaluated in Chapter 5.

[57] Shared, writable scalars such as the generation counter usually benefit from being padded so as to preclude *false sharing*. CPU cache lines nowadays are 128 or, more commonly, 64 bytes. Absent complier alignment directives, padding a 4-byte integer into a dedicated cache line entails placing inert, dummy entities of 60 (or 124) bytes on both sides of the

One method, used in the tested implementation of *agree*, deploys the entire scratch slot array on the stack. For each unification operation, an initialization function subtracts a value from the stack pointer, allocating a block of memory in the topmost frame of the unifying thread's stack. The entire scratch buffer is organized in this buffer as a contiguous array. Stack-based allocation eliminates a costly system allocation cycle per unification and trivially gives each unifying thread a private set of scratch slots, eliminating the need for centralized coordination of scratch slot leasing. Naturally, with stack-based allocation, the entire set of scratch slots is instantly discarded, in the case of unification failure, with a single subtraction instruction. Indeed, this subtraction is implicit and automatic upon execution returning to the caller.

The first call which starts the recursive unification, proper, is initiated from within the initialization function, so pointers to the scratch buffer are accessible to, and valid throughout, any call below the initialization function. These frames access these fields via direct pointers to the initialization frame. The buffer automatically disappears only when the unification is complete.

A disadvantage of this method is that the second pass—the *realization* or writing pass, which traverses the scratch slots to produce a result TFS—cannot be deferred, a technique used in "hyper-active" parsing (Oepen et al. 2000). This is because there is no way to prevent the loss of the stack frame when the unification function returns to its caller. Also, when we root out exactly where the task performed by the generation counter 'went,' we find that the function of the generation counter is simply hidden: security considerations in the runtime platform mean that it is not possible to opt-out of the zeroing-out of the memory block that is allocated from the stack as described. Although the runtime environment probably effects the clearing with a fast block-storage processor instruction—or perhaps even an outboard direct memory access (DMA) executor—the courtesy surely comes at some cost.

The reason I detail this point is that the array storage unifier is substituting the function formerly performed by the generation counter with this runtime guarantee. The generation counter is not needed in array storage because each unification operation gets a fresh set of scratch slots which can never be contaminated with abandoned values. Because memory-clearing applies to every type of allocation, it is difficult to avoid this platform-dependent penalty without explicitly managing long-term buffers—as van Lohuizen does. The trade-off is that his method invites contention where the stack method incurs none. Finally, note that zeroing a stack allocation is not automatically performed in C/C++, so adopting stack-based allocation and eliminating the generation counter would require some kind of explicit initialization pass over the slots, in order to distinguish their first use from randomly occurring values.

Recent work (not reflected in the results presented in Chapter 5) has altered the scratch slot allocation method used in *agree*, and the generation counter method introduced by Wroblewski

---

oft-written field. These guarantee that no shared, read-only fields are placed within the volatile cache line, where they would be needlessly incur penalizing cache misses.

(1987) is now used. To permit concurrent unification, a platform mechanism called *thread static storage* automatically associates—via a single object reference—a complete and independent unifier implementation with each thread. Each of these unifiers has its own private set of reusable scratch fields governed by a private generation counter. It is impossible for these resources to be contended-for, because they are contained within the 'unifier' object, and this is always known to be privately owned. The operating system provides and efficient guarantee that each access to—what appears to be—a singleton, thread static 'unifier' object by each different CPU actually returns a private object invisible to the other threads. A small number of dedicated unifiers quickly become available for each physical CPU in the system, and remain active for the lifetime of the *agree* process, and this minimizes the costs of reinitializing the fixed set of resources used by each instance. In turn, by reusing a single scratch slot array throughout its own lifetime, each unifier is able to avoid expensive memory-zeroing penalties after each operation.

4.4.4 Scratch slot implementation

This section describes the scratch slot implementation used in the array TFS unifier. Each scratch slot has the form

$$_{(Type,\mathbb{Z},\mathbb{Z},\mathbb{Z})}\langle t, m_T, ix_{fwd}, m_{copy}\rangle. \tag{4.9}$$

Not shown is an additional field related to node counting: the calculation of the total number of nodes in a newly-unified TFS. Node counting is required for pre-allocating an empty array $\mathbb{A}$ of the exact required size, prior to the writing pass in a successful unification.[58] A detailed presentation of this topic is not provided in this thesis, but a summary of the salient points is provided in Section 4.4.10. Note the absence of a COMP-ARC-LIST field; as described in Section 4.4.1, after initialization the array storage unifier manipulates no operationally variant data structures, and is always able to completely describe the result structure via the ix_{fwd} field alone. With regard to this claim, accounting for the other three fields in (4.9) is simple. t and m_T are copies of the *out*-tuple. With the exception of a trivial optimization (described in Section 4.5 on page 97) which modifies t but is substantially unimportant, these fields remain unchanged and can be interpreted as being related to performance-related caching. The m_{copy} field pertains to the writing pass, which represents a trivial re-expression of the result structure which has already been fully described by the unification proper. It is also worth mentioning that, as is the case with any scratch slot traversal, reference to the set of read-only arity-maps of the input arguments is also needed to read-out the result structure, but these are also not counted as a unifier work product since they remain unchanged since before the operation began.

To be completely precise, the set of all ix_{fwd} after unification is not quite complete, because a starting slot value where the read-out of the result begins must also be specified. For example,

[58] In the target platform, instances of the array primitive, the lone vehicle capable of providing adequate performance for this application, cannot be resized after creation.

in the case of mother-daughter unification, this will be unifier slot index 1, which corresponds (in the latest *agree* slot layout) the root node of the mother argument TFS. This is because the topmost node of the result of a mother-daughter unification will always have the type of the mother. Also, of course, no daughter structure is available to choose from here.

The stack-based allocation described in the previous section, considered as a single array of scratch slots, is demarcated into adjacent regions of contiguous scratch slots dedicated to each TFS that participates in the unification. Within each of these regions, there are three sub-regions. First, the unifier designates a special set of slots equal to the number of coreferenced nodes in the TFS. As noted in Section 4.4.2, rather than trying to choose one amongst the multiple scratch slots which may be reported for a single coreferenced node, in this design the unifier ignores all array storage indexes selected by negative *in*-marks. Taken together, the negative marks for the set of coreferenced nodes in a TFS comprise a set of consecutive negative integers descending from -1, so no mapping is required. Any node with a negative mark is redirected to one of these special slots by using—instead of the node's (non-unique) slot index—the actual value of its mark itself to select (as a negative offset) its scratch slot, and this redirection effects the conflation of coreferenced nodes.

Immediately following the special coreferencing singletons, a single scratch slot is designated for the TFS root node. Because the root *out*-tuple is not stored in $\mathbb{A}$, its array storage index is undefined, and therefore requires special treatment. Additional discussion of this point can be found in Section 3.2.1. Finally, a set of scratch slots, equal in number to the size of the array TFS storage, is created for non-coreferenced nodes. Note that there may be numerous scratch slots in this sub-region which go unused, due to the redirection of coreferenced nodes to their own special sub-region. These 'holes' are a benign condition.

The unifier operates according to the dereference-before-forward principle from UNION-FIND and most later unifiers (but not *n*-way unification). As with all methods based on the technique, the ix_{fwd} (forwarding) field is perhaps the most important. Forwarding between scratch slots observes a system which spans across the entire scratch array, encompassing all participating TFSes. Specifically, ix_{fwd} is an integer which represents a global index of a scratch slot, or zero if the slot is not forwarded.[59] This permits any scratch slot to be forwarded to any other, within the same, or any other, TFS.

Since coreferenced nodes use their (negative) mark to directly select a scratch slot, it is worth describing why this technique is not used for all nodes. The answer is that, in the current design, nodes whose type has no appropriate features are always given *out*-mark value zero. Because of this, unilaterally using a node's *out*-mark as a scratch slot index would incorrectly conflate all nodes of the same type across the TFS. The design whereby a zero mark value is

[59] The first scratch slot, at global index zero, is reserved and not issued to any argument TFS.

given to featureless nodes is a legacy from early in the evolution of this project whose original motivation has been deprecated.

Within the scratch slot region for a given TFS, a single base address provides access to all of the TFS's scratch slots. From the start of the TFS's region, this base address is given a positive (or zero) displacement of $-\min_{m_T} A$, so as to permit direct indexing in the inclusive range

$$\left[\min_{m_T} A\,,\,|A|\right]. \tag{4.10}$$

The displacement allows negative mark values, which indicate coreferenced nodes, to directly index scratch slots in the special coreferencing sub-region. The special scratch slot for the root node is at offset zero from the base. Non-coreferenced nodes add one to the node's zero-based array storage index to directly access slots in the range $1 \ldots |A|$. During unification, a special array storage access mode is used to request that the array storage index, which is a free by-product of the access, be returned along with each node access. Naturally, this index, along with the node's type and *out*-mark, is only accessible during the feature-value enumeration performed by the caller. To summarize, for each argument node, three pieces of information are maintained: the array index of the governing *out*-tuple in the argument TFS, and, from the tuple itself, the type and *out*-mark.

To process a substructure node, only a single pointer is passed down in the recursive call. This is a pointer directly to a single scratch slot, calculated from the TFS's base address and an offset as described above. The slot index, one of the three items, is incorporated into the pointer itself, of course. For efficiency, the other two values which must be passed down are cached within the slot. Referring to (4.9), these are t, the type of the governing node, and m_T, the *out*-mark of the governing node. The latter is combined with features appropriate to the former, each in turn, to select follow-on nodes. Caching the node type in the slot allows the slot to be more flexibly used with the fallback fetch mechanism described in Section 4.4.8. Caching of *out*-marks is also helpful because they mediate the accessibility provided by the mappings co-opted from TFS storage, as described in Section 4.4.7 and are thus referenced frequently during traversal.[60] From a formal standpoint, however, caching the *out*-tuple in the unifier slot is not strictly necessary.

At this point, the alert reader may quibble that caching contentful values (such as the node type and *out*-mark) in the scratch slots amounts to a form of over-copying, denounced in Section 4.2. In fact, however, the presence of these values in a persistent portion of the stack frame is little different from their presence as arguments to recursive function calls. Function calls involve stacking and unstacking numerous values. While acknowledging that the control triple discussed here contains non-opaque internal structure—and is physically wider—than a simple, opaque node pointer, it should nevertheless not be considered "copying" to pass control

[60] They also allow an extra TFS access to be avoided when establishing the fallback condition, but this point is not detailed further here.

information to a lower stack frame. The optimization here is simply to consolidate the multiple control values under a single pointer. This pointer happens to refer to an entity—an upper-frame scratch slot—that is more persistent than a conventional function argument. The savings of consolidating arguments and operating exclusively on scratch slot pointers is that slots which undergo multiple operations during the course of an operation are prepared only once.

4.4.5 Array TFS storage slot mappings

Sections 4.4.1 and 4.4.2 described *how* array storage layouts are co-opted by the unifier for the purpose of organizing a set of scratch slots, but elided the issue of *why* these layouts have value in the first place. Surely the unifier could, on its own, arrange an appropriate configuration of slots. The value of the storage layouts lies in the fact that each is a pre-made arity map for every 4-tuple in $\mathbb{A}$, essentially a feature-arity configuration customized to each TFS. Each mapping provides slots for the constrained features associated with every node. Taken by node, each of these subsets can be chosen as a standalone *slot-mapping* for that set of features. For each argument TFS, the proper number and type of slot mappings are already laid out, which means that each has a complete schema that is fully established and available prior to starting the unification operation.

A slot mapping is defined as the subset of slots (represented as slot indexes relative to some array storage TFS) that are obtained with a particular *out*-mark value (again, from that TFS). This is the familiar set $\sigma_{m_F=m}(\mathbb{A})$, augmented now with (notational) array storage index values.

Slot mappings, which are each local to their TFS, become useful to the unifier because for each argument TFS F_i it assigns a global range of unifier scratch slots to the entire storage relation $\mathbb{A}_{F_i}$ (in simple one-to-one correspondence with the local index values), thus trivially and instantly relating all of F_i's local maps into the range. Scratch slots can be globally forwarded, crossing local TFS boundaries, enabling the unifier is able to assemble a patchwork from the incorporated maps which ultimately describes the unification result TFS.

To summarize, array TFS storage internal layouts are valuable because they represent the work of compacting a set of mixed-arity feature-value data—the structure of a particular argument TFS—into a ready-made hash table. While there is no way to alter the slot mappings that this hash table subsumes, the unifier is free to choose a slot mapping from any input TFS that offers one which is associated with type of the current node. Choosing the representative slot amounts to choosing the best m_F where the set of scratch slots accessed by $\langle f, m_F \rangle$ is fixed.

4.4.6 Slot mapping selection

In this section, I show how the array storage unifier constructs the result TFS by forming a strategic patchwork of pre-made slot mappings.

The set of slots in a slot mapping is organized under a single *governing slot*. During the processing of feature-value pairs, this is the view looking "upwards" (i.e. towards the TFS root, backwards over structure just traversed). Looking "downwards" (i.e. towards the next iteration, forward away from the TFS root) invokes a different sense: for each feature-value pair, a *representative slot* is chosen, based on the slot mapping it offers, over other available slot mappings. The representative slot becomes the governing slot for the node's substructure. At any given time, then, there is a single governing slot, and the task at hand is to iterate over feature-value pairs, choosing representative slots.

It is important to realize that the choice of slot mapping does not alter the possibility of choosing representative slots from either input argument—or a mixture from both—amongst the joint set of feature-value pairs. Consider how a governing scratch slot denotes some list of features beyond those referenced by its corresponding slot mapping. In grammars with a feature-appropriateness condition, this might mean switching the slot's current type to a (more constrained) subtype, so that a larger set of features becomes appropriate. Conceptually, increasing the set of accessible features is a simple matter of changing the slot's type. Recall from Section 2.3.6 that, in array storage, doing so results in the instantaneous appearance of unconstrained nodes $\langle f, \top \rangle$ for the enlarged feature set; we have expanded the *accessibility* that the mapping provides. This does not suggest a solution for the current problem, however, because these "chimeric" nodes, being synthetic, have no storage index and hence no scratch slot of their own.

Nevertheless, the preceding example is instructive, because it illustrates that the choice of representative slot only establishes the *accessibility* (or governance) of a set of scratch slots via some *out*-mark. Some of these—or all, or none—may be designated as representatives (versus being forwarded to a representative) themselves. But beyond providing access to a fixed grouping of scratch slots, the mapping constrains nothing. The task of the unifier is to designate, as governing slots, a subset of the predefined, fixed set of slot mappings offered by the argument TFSes, and arrange them such that the resulting accessible substructure corresponds to the result TFS.

Any mapping that provides slots for the set union of constrained features between the arguments can be selected—regardless of the contents of nodes it selects in its argument TFS. Those contents never affect the choice of mapping because the type of any unifier scratch slot can be arbitrarily changed, and each slot selected by the chosen mapping can be forwarded to reflect the necessary substructure. Therefore, the choice of mapping is based solely on the feature accessibility offered by the mapping.

A problem case arises when none of the available slot mappings have an *out*-mark which offers slots for the complete set of required features. This is the disjoint coverage problem, discussed in the next section.

4.4.7 Disjoint feature coverage

To summarize the issue and the discussion thus far, the task of the unifier is to construct an arrangement of slot mappings that makes a unique scratch slot *accessible* for every node required by the result feature structure. We recalled that for attaching data to a set of variable-arity features, array TFS storage uses the technique of predicating TFS node extraction upon the grammar's feature-appropriateness condition. But this does work for the unifier's problem; since slot mappings are co-opted from array storage layouts, they do not report valid slot indexes when queried with features newly appropriate to the subsumed type, or which were not stored for any other reason. Although we can expand the accessibility that a mapping provides, we cannot give it slots that it did not originally have.

Array TFS unification begins by reducing the frequency of the problem in two ways. First, because unification is commutative, cases of disjoint feature coverage can be further reduced by unifying-in the well-formedness constraint *prior to* unifying the subsuming argument nodes. For example, if $q.t \sqcap r.t = g$ and the root node of canonical constraint TFS G is node $\bar{s}$, then unification can proceed according to $q \sqcap \bar{s} \sqcap r$, as opposed to $q \sqcap r \sqcap \bar{s}$. Because

$$\begin{aligned} \left|\bigcup_{f \in \textsc{Feat}, \mathcal{A}pprop(f,q.t)\downarrow}\right| &\leq \left|\bigcup_{f \in \textsc{Feat}, \mathcal{A}pprop(f,g)\downarrow}\right| \text{ and} \\ \left|\bigcup_{f \in \textsc{Feat}, \mathcal{A}pprop(f,r.t)\downarrow}\right| &\leq \left|\bigcup_{f \in \textsc{Feat}, \mathcal{A}pprop(f,g)\downarrow}\right| , \end{aligned} \tag{4.11}$$

$\bar{s}$ is most likely to express the greatest coverage of features expressed at the result node.

Second, the decision of which slot to select as the representative is made only after examining which of them provides the slot mapping with the most coverage. By tallying the number of features found in q but not r—and vice-versa—the slot mapping which provides the greater coverage is always selected.[61] Note that we evaluate each slot based on how well it would work in role of *governing* its substructure, even though here we are selecting the representative slot. After being selected, the type stored in the representative slot is updated to reflect the running type unification result. This is the current type for the equivalence class represented by the joined scratch slots, and this type always carries precedence over the type values in forwarded slots or the type values in their original argument TFSes.

If for every node q, array storage physically stored a value for every feature that is appropriate for $q.t$, then the two techniques detailed above would be sufficient to guarantee that at least one of the two nodes being unified would have an equal- or superset of the other's features. A slot mapping which covers the maximal set of appropriate features at each node would always be found. This is not necessarily the case in the array storage design described in this thesis, where unconstrained feature-value pairs $\langle f, \top \rangle$ are stored only when they are coreferenced (but

[61] This implies that the node forwarding decision is deferred—and no forwarding enacted—until outbound from depth-first traversal. Tomabechi (citing Marie Boyle at the University of Tuebingen) notes that this deferral, which was in effect in his 1992 algorithm, is incompatible with the successful unification of cyclic structures, so he switches to inbound forwarding in his 1993 method. The consideration does not affect the current work, which assumes a formalism that prohibits cyclic structures.

see footnote 37 on page 46). This allows the disjoint coverage problem to persist in rare cases, as it is possible for arbitrary scratch slots to be missing by virtue of one of the input nodes leaving a feature neither constrained nor coreferenced.

4.4.8 Fallback fetch

This section continues the discussion of the solution to the problem of insufficient coverage amongst the available feature maps at a given node during unification. Because the motivating problem has been obviated by recent modifications, the method described in this section is no longer used by *agree*, but the section is retained since it accurately describes the implementation which is evaluated in Chapter 5. The more recent solution is summarized in footnote 37 on page 46, and proceeding directly to Section 4.4.9 will not significantly disrupt the narrative.

Disjoint coverage can occur either because the TFS from which the mapping was co-opted did not store a constraint for the feature, or because the type in the mapping's scratch slot has been changed to a subtype of the value that was originally present in the source TFS. The mitigation techniques described above reduce instances of the problem but do not eliminate it entirely. To address the remaining cases, a mechanism dubbed *fallback fetch* is used.

The method follows from the observation that every feature that ends up needing representation in governing node's set of features—because it is constrained—must ultimately originate from one of the input structures. What is needed is a way to designate one of the parent slots for that node as an alternate to the selected governing slot. Accordingly, in the array storage unifier, the slot forwarding mechanism is enhanced so as to permit an alternate forwarding condition to be signaled, if necessary, and this feature is activated whenever the best available slot mapping is missing a slot index for one or more features that are appropriate to its type. After being set for a slot, the fallback fetch condition asserts that, for any features that are found to lack coverage in the primary mapping, all future feature queries will query one or more backup slot mappings.

In short, the array storage unifier provides the ability for a representative slot to signal that zero or more additional mappings be used as backup slot mappings. The mechanism is only activated by the unifier in the rare case when the representative slot, which has been chosen so as to maximize its feature coverage, is found to lack coverage for a feature for which one of the rejected mappings supplies a constraint.

Note that it is not the slot of the disjoint constraint that is configured for fallback. The reason goes back to *accessibility*: fallback allows extra *coverage* to be *discovered*—which is preliminary to *retrieving* additional content. Ultimately, fallback fills in gaps in *feature* coverage, but it must always be configured between the feature's *governing* slot (and its alternates).

The engineering implementation of fallback fetch is described next. Note that the set of slots that need to be related by fallback fetch always meet the criterion that they are all already for-

warded to the primary slot. For this reason, the implementation overloads the forwarding mechanism to signal the condition. To activate fallback, the forwarding chain is configured as a loop—rather than a singly-linked list—with the governing slot specially marked as the loop *master*. Note that only those mappings that need to supply fallback constraints for one or more of their subjugate feature-value pairs need to join the fallback loop. Additional singly-linked chains of slot mappings whose features were fully covered can still lead to the governing slot with normal one-way forwarding chains.

By default, fallback is disabled, and a forwarding chain is followed until reaching a terminal scratch slot, respecting the dereference-before-forward procedure as normal. When activated for a governing slot, fallback fetch works as follows. When any query (for an appropriate feature) returns $\langle \mathsf{T}, 0 \rangle$, the loop is followed to an alternate scratch slot. This scratch slot represents an alternate mapping: it contains the mark of some alternate node—and, of course, which of the participating TFSes that node is in. This allows the unifier to re-query a (possibly) different TFS with an alternate mark value, and the same feature f. If successful, this operation returns an array storage index (or negative mark value) which selects a scratch slot which controls the substructure below f (or is forwarded to the same). In the extremely unlikely event that the first alternate mapping does not cover the feature, the process continues with additional alternates. If no alternates respond—meaning that the loop was followed back to the master governing slot—then the node is known to be truly unconstrained.

4.4.9 Summary of the first pass

The material contributions of the array storage unifier are confined to the first pass of the array storage unifier; the writing pass proceeds in a conventional way. Accordingly, before proceeding to a description of the writing pass, I briefly review Sections 4.4.2 to 4.4.8, highlighting just the areas where the operation of the array storage unifier differs from previously published work.

As in earlier methods, the new method associates a scratch slot with each node. The slot contains a forwarding pointer which can indicate any other scratch slot (or none) belonging to any other TFS that is participating in the unification operation. Each slot also contains a current type, and the information necessary to access its array TFS source. These last two items are not formally required, and are cached for performance improvement only. Each scratch slot contains, in total, six scalar values (one is used in the realization pass, and one is related to node counting, described in the next section), and only the forwarding pointer is logically list-valued (though physically scalar). Naturally, the lists implemented by forwarding—being singly-linked lists (or singly-linked loops contained to the active scratch slots in the case of fallback fetch)—are allocation-free. To enable concurrent access, scratch slots are detached from the TFS by conscripting array storage indexes as indexes into a contiguous array of slots.

Unification proceeds according to UNION-FIND. At each node, the unifier ensures that it chooses, as the representative, the slot mapping which covers the most features. Only actual cover-

age is evaluated for this decision. The unifier also enacts the well-formedness unification, if required, on one of the input nodes prior to unifying the actual arguments. This increases the likelihood of a complete coverage map being found—in the well-formedness TFS. Both of these techniques are optional, but they reduce the frequency with which fallback fetch must be established, and fallback fetch is expensive.[62]

After mitigation, if the representative slot provides full coverage, then normal, one-way forwarding is established from the non-chosen slot to the representative. Despite mitigation, however, incomplete coverage can still persist, owing to the array storage design where non-coreferenced `T` nodes are never stored. In this case, a special forwarding relationship is established between the "master" governing slot and one or more alternates, to signal that an alternate mapping must be consulted whenever the master slot fails to cover any feature appropriate to its type. This incurs extra lookups, but by having selected the slot which provides the most coverage, these cases are rare. The special forwarding relationship, the presence of which signals the need for the fallback lookup, is that the master's forwarding pointer is non-zero, and in fact, is part of a forwarding loop.

4.4.10 Node counting

Prior to writing any new structure, the array storage unifier must allocate a fixed-size array to accommodate the tabular storage. This entails that the number of nodes in the new structure be known after the initial unification pass. During the first pass, a mechanism not described in detail in this thesis monitors how many nodes the result structure will contain. Unfortunately, it is not trivial to determine the number of nodes in a result structure when the unifier has skipped over sections of substructure due to the application of feature restriction or rule-daughter truncation.[63] Furthermore, coreferencing can join structures containing substructures that were previously joined, and those portions must not be counted twice when determining how many nodes to subtract from the running total. Another complication is that nearly all parsing unifications involve three entry points (the tops of the candidate and mother TFSes, respectively, plus the daughter `ARG` position within the mother). Obviously two of these are within the same TFS, and this should not lead to double-counting.

But most problematically, the seemingly harmless array storage design—wherein *out*-tuple $\langle \mathtt{T}, 0 \rangle$ is never stored—creates (at least) two additional complications: first, merely applying a

[62] Avoiding the activation of fallback fetch is desirable because it is a blunt instrument: if needed due to the lack of coverage for just some single feature, it must be activated on the governing (i.e. mother) slot and thus affects all of node's features; even those features which none of the argument TFSes care to constrain will be exhaustively queried.

[63] Note that the unifier's introduction of additional TFSes for the purpose of enforcing well-formedness is not necessarily a category of problem here, since total *visited* nodes are easily counted; see Section 4.4.10. In general, the problem occurs because an efficient unifier will not bother—in its first pass—to visit below any structure that only one of the input structures constrains. Static analysis of the input structures does not help: the amount of restriction in a result TFS is not related to the (invariant) amount of the restriction in the input TFSes. To see this, consider rule-daughter pruning: any coreference in a rule (mother) that spans both above and below the daughter entry (i.e. ARGS) allows the daughter to join—and/or publish to the result TFS—arbitrary sections of the mother's substructure, and these sections might contain nodes that are subject to restriction, while not being subject to pass 1 visitation.

type to an unconstrained node—even if the new type has no appropriate features—requires the node count to be incremented. Failing to account for this case results in undercounting, which is fatal because the realization pass will run out of array storage entries.

Second, when structures are joined such that existing coreferences in their substructure become vacuous, and when these newly non-coreferenced nodes also happen to be unconstrained, the total result node count must be *decremented*—because now a result node will not be stored. For example, consider the unification of feature structures C and D in Figure 15. In the result TFS E, the coreference between paths $G.F$ and $H.F$ established by TFS C has become vacuous because the two paths are no longer distinct. The issue is important in *agree*, where every node carries a bit that indicates whether or not it is coreferenced. Any error in the correctness of this bit is considered unrecoverable corruption.

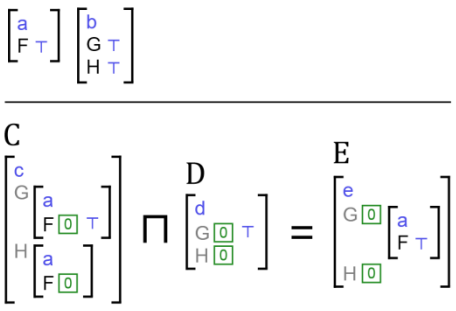


Figure 15. Coreferences can become vacuous during the course of unification. When TFSes ***C*** and ***D*** are joined giving ***E*****,** the reentrancy that was originally authored in ***C*** becomes moot since paths ***G*.*F*** and ***H*.*F*** refer to the same node.

For the unifier's application of node counting, this very rare case—which results in overcounting—can safely be ignored. The benign but unaesthetic result is that storage relation $\mathbb{A}$ ends up with one or two extra unused rows—out of several hundred—in some TFSes.

For these and other reasons (primarily related to feature restriction), node counting is a challenging aspect of the array storage unifier. It is also the case that node counting worsens the theoretical complexity of the unifier, because it requires pursuing unary descent—to count nodes below singleton nodes in an input argument—in some cases. Absent node counting, the unifier could simply ensure that unary substructures are accessible in the governing slot mapping, and skip over the node, without descent.

Whether by explicit counting or some other method, the array storage unifier requires—after the completion of the first pass—some estimate of the (maximum possible) size of the result structure. In the target runtime environment, arrays, once allocated, cannot be resized, and realization cannot begin until the destination array has been allocated. Therefore, without node counting during the first pass, the implementation would require three-passes; an additional traversal of the entire result structure (as manifested in the scratch slots) would be needed in-between the first and second passes. As noted throughout Section 4.4 and in footnote 37, recent work in array TFS storage has modified the design so that the storage of unconstrained, non-coreferenced nodes is no longer prohibited. Eliminating the special treatment of this case has enabled several simplifications, but as the discussion of this section has shown, accurately predicting the number of nodes in a unifier result structure is one area where the potential for simplification is considerable. At this time, examination of this problem is an area of very active investigation.

4.4.11 Writing pass

If the joint traversal completes successfully, unification success is guaranteed. Having reached this point, the result structure is encoded across the scratch slots, in a configuration of interwoven slot mappings borrowed from the input TFSes. The second pass of the array storage unifier writes the final TFS. The primary task is to copy data from the scratch fields into the rows of a new array storage relation, and allocate a system object A to bind together, by containment, the sundry chunks of data that comprise the feature structure. Most significantly, before the writing pass can begin, a persistent, 4-tuple array $\mathbb{A}$ of the required size, as determined by node counting, is allocated. Also initialized is the root *out*-tuple, which, in accordance with (2.33) and Section 3.2.1, is maintained external to the 4-tuple relation. This standalone *out*-tuple is created with the proper type and is given *out*-mark value 1.[64] Two sequence counters, each incremented away from zero, are initialized for assigning marks. The sequence for coreferenced nodes starts at -1 and the sequence for non-coreferenced nodes starts at 2.

The writing routine is a simple traversal of the graph embedded in the scratch slot tapestry. If configured, the fallback fetch condition is respected when navigating slot mappings. Entering at the root, the algorithm traverses the structure in full, following forwarding chains to their terminus when they are encountered, arriving at a representative slot for each node. The type stored in this scratch slot is the final type to store as an *out*-tuple.

Note that the terminal slot in a forwarding chain determines only the *out*-tuple that is recorded. Although the representative slot which we arrived at—like all scratch slots—appears in one or more feature-arity maps of its corresponding TFS, these are irrelevant to what we are now recording.[65] The correct feature identifier to emit as the *in*-tuple for the current node is the feature value, appropriate to the node type, that we originally queried from the governing slot, prior to following any forwarding.

The representative slot stores information about how to proceed into its substructure, if any. This is the slot mapping, which consists of a reference to one of the argument TFSes (implicitly, via its global slot index), and an *out*-mark to be used with its hash addressing scheme. This mark is paired, in turn, with each feature appropriate to the result node type to obtain a set of storage indexes relative the indicated TFS. These TFS-local indexes are trivially mapped into the range of scratch slots assigned to the TFS, thus designating a set of unifier scratch slots. If forwarded, these are followed as the process is repeated.

---

[64] By using a positive value, the system is asserting that root nodes are never coreferenced, which must be the case since cycles are not permitted in the adopted formalism. As with the *out*-tuples resident in $\mathbb{A}$, the root *out*-tuple is described by a C# value type, so it would not be precisely correct to say that the root *out*-tuple is itself "allocated." Its physical storage exists *in situ* (along with a handle (reference) to the C# array which implements $\mathbb{A}$) within A, the object that represents the array TFS storage instance as a whole.

[65] Recall that an *in*-tuple records the 'arc,' or the identity of the last feature in the path to the node.

During the first pass, in addition to determining the total node count, the node counting mechanism records, for each scratch slot, how many feature-value tuples refer to the slot.[66] If there is only one, the result node is not coreferenced. In this case, the next positive mark value is issued, unless the node type has no appropriate features, in which case mark zero is used. If there is more than one governing slot which references the node, the result node is coreferenced, and the next mark in the negative sequence is issued. For coreferenced nodes, the mark value is also placed in the scratch slot's m_{src} field, so that later references to the node will use the same mark value, which establishes coreferencing in the array storage relation. When the value for each of the four fields in a 4-tuple have been gathered, the tuple is written to the current position in $\mathbb{A}$, and the writing position is advanced.

4.5 Example

This section presents step-by-step diagrams which depict the operation of the unifier at each stage of a unification operation. The section begins with some explanatory remarks to assist with the interpretation of the diagrams.

Recently, the method by which the unifier assigns scratch slots to the coreferenced nodes of the argument TFSes has been altered to capitalize on the forwarding mechanism used by the unification procedure itself. In short, the distinct range of scratch slots dedicated to the coreferences of each argument structure (as described in Section 4.4.2) is replaced with a simple array of pointers to unifier slots. The slot indicated for a given coreference becomes its 'designated' slot, a condition trivially claimed by the first node the unifier encounters from each equivalence class. Henceforth, each node from the corresponding set of joined argument nodes is forwarded—using unifier's own forwarding mechanism—to the designated slot (or according to its forwarding chain). This enhancement will be described in more detail in future work, but is mentioned briefly here because the diagrams shown in this section reflect the modified design. One motivation for this is that illustrative diagrams are more concise and compact when the number of scratch slots is fewer.

In *agree*, a single "top-level" TFS unification operation[67] can incorporate any number of input arguments, called unification *participants*. At least one of the participants must be designated as denoting the outermost root of the putative result TFS, and each of the remaining participants is unified into a particular substructure position (or the root) relative to the root of that argument. Additional participants can be introduced into the operation at any time, and this

[66] As was the case for *n*-way unification (and noted in footnote 52 on page 61), the criteria here is not the number of referencing *paths*, but rather just the number immediate parent 'arcs.' In terms of unifier scratch slots, this notion of immediate parents is subtle. We wish to mark as coreferenced any slot which is reached as a forwarding target more than once, but this determination is unrelated to whether the target slot is itself reachable, that is—via some slot mapping within its own TFS. In fact, it is very common for slots representing arcs that no longer contribute to the result structure to nevertheless remain as active governing slots which maintain a type and mark value for a result node.

[67] By "unification operation," I refer to (re-)setting the number of input participants to zero, and (re-)initializing the unifier's scratch slots. In the most recent implementation and as described in Wroblewski (1987), the latter amounts to incrementing a single integer value which serves as the unifier's generation counter.

allows well-formedness TFSes to be incorporated, as needed, without the expense of starting an entirely new operation. For mother-daughter unifications, this also allows an entire set of rule daughters to be unified as a single operation, an optimization that showed significant benefit when evaluated with *n*-way unification (see Figure 13 on page 72).

The example that will be used to illustrate the operation of the array storage unifier is the mother-daughter unification shown in Figure 16. The operation involves two TFS participants—M, the mother, or *outer* argument, and D, the daughter, or *inner* argument. As noted above, additional participants can be introduced, but if they do not coincide with the outermost structure, they all must be positioned relative to (i.e., within) the same outer participant, which must be specially designated as such.

Overall, there are two basic steps. First, D is unified into a particular sub-structural position within M. This step is the only part of the overall unification that is subject to failure, so it is evaluated directly first, making no reference to any parts of the mother that fall outside the reach of the daughter. No traversal is required to locate the daughter entry points of the outer structure, because each mother-daughter (i.e. "rule") TFS caches all of its own daughter marks (and slot indexes, for the unifier's benefit). If the first step is successful, the second step builds a result structure by incorporating the unification result—as manifested across the unifier's scratch slots—into a copy of the untouched portions of the outer structure.

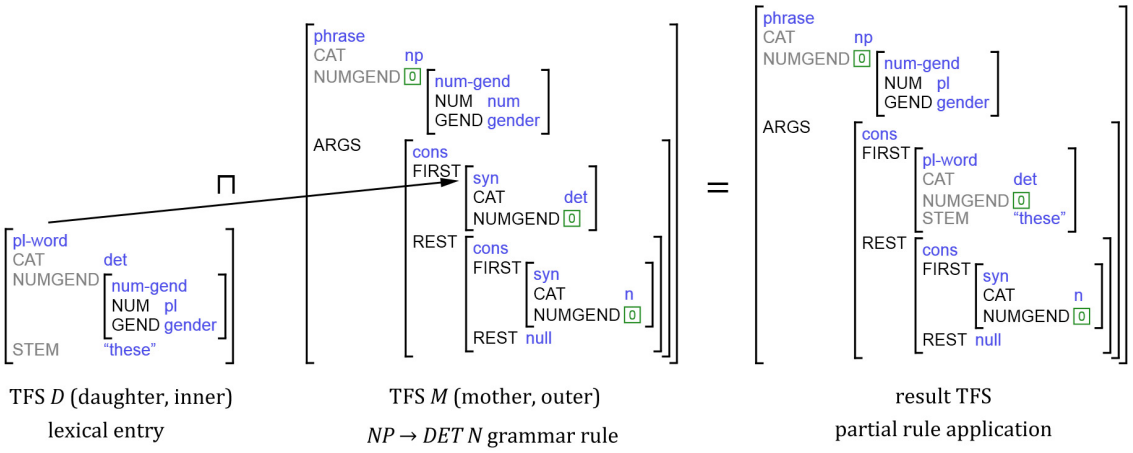
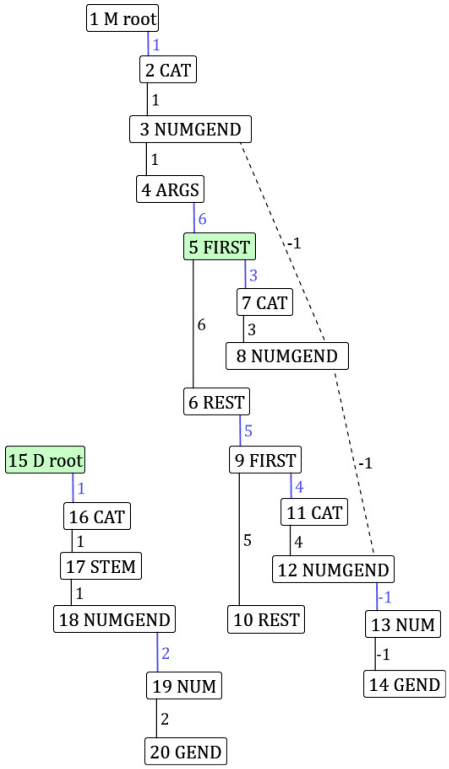


Figure 16. The TFS unification used as an example in this section.

The sequence of examples each correlate tabular values from the unifier scratch slots with the same information presented in the graphical form introduced in Section 2.3.4, a new representation suited to the modeling of the invariant arity mappings that are exposed by array storage TFSes. The use of these non-directed graphs emphasizes an important feature of the array storage unifier, namely, that it expresses the result structure amongst its scratch slots without having to alter the given slot mapping substrate. Figure 17 shows the state of the unifier's scratch slots after the two participant TFSes have been introduced but prior to the start of unification proper. The main array of scratch slots is labeled **ps_base**. Here, the entire set of storage rows

for each participant TFS, plus one slot for each root node, have been mapped into a set of contiguous scratch slots. This mapping is instantaneously accomplished by simply asserting 1-to-1 correspondence between the (storage) ordering of every tuple in each participating $\mathbb{A}$ and some range of unifier slots. In other words, each TFS is given a unique, abutting range of slots.



ps_base, step 0

slot	TFS	m_F	f	gen	ix_{fwd}	t	m_T
0	(global slot 0)			unused			
1	M	(root slot M)		old			
2	M	1	CAT	old			
3	M	1	NUMGEND	old			
4	M	1	ARGS	old			
5	M	6	FIRST		0	syn	3
6	M	6	REST	old			
7	M	3	CAT	old			
8	M	3	NUMGEND	old			
9	M	5	FIRST	old			
10	M	5	REST	old			
11	M	4	CAT	old			
12	M	4	NUMGEND	old			
13	M	-1	NUM	old			
14	M	-1	GEND	old			
15	D	(root slot D)			0	pl-word	1
16	D	1	CAT	old			
17	D	1	STEM	old			
18	D	1	NUMGEND	old			
19	D	2	NUM	old			
20	D	2	GEND	old			

pp_corefs

TFS	coref	slot
M	-1	(null)

Figure 17. The state of the unifier slots after preparing for the mother-daughter unification shown in Figure 16 and prior to initiating the unification traversal. Two scratch slots have been initialized: the slot corresponding to the root node of the daughter, and the slot for the daughter's position within the mother. These are the slots which will be joined first, in a call to the recursive procedure that unifies two TFSes, node by node. The remaining scratch slots are untouched, as indicated by an 'old' generation value. See the text for further discussion.

The first four columns in **ps_base** have no manifestation in the scratch slots, and their contents—as mediated by the participant mapping—are provided only to facilitate this presentation. One of the remaining four columns, 'gen,' implements a generation counter (Wroblewski 1987) that allows an entire set of unifier slots to be efficiently re-initialized (for reuse in a subsequent operation). This mechanism substitutes for the stack-based slot allocation method, which was also described Section 4.4.3 (as noted, this recent aspect of the *agree* implementation is not incorporated in the system evaluated in Chapter 5). For any slot that belongs to an 'old' generation, the table will show no further information, because the slot has not been touched by the current unification operation, so the values in those fields are unpredictable.

The three remaining columns are described by (4.9) in Section 4.4.3. To review, ix_{fwd} contains the slot's forwarding value, an integer which indicates the index of any other unifier slot, allowing slots to be globally forwarded across participant boundaries. A value of zero indicates

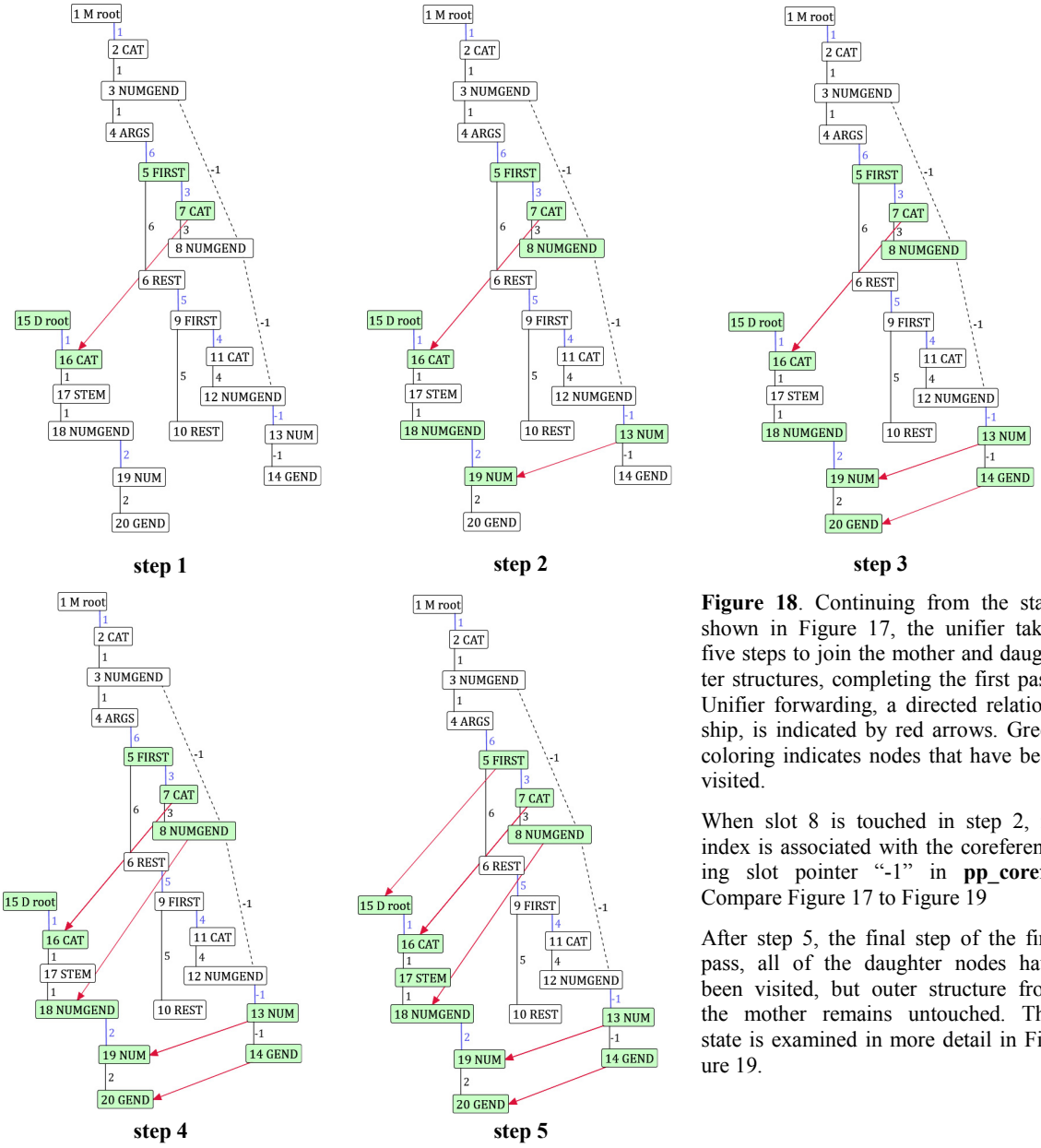


Figure 18. Continuing from the state shown in Figure 17, the unifier takes five steps to join the mother and daughter structures, completing the first pass. Unifier forwarding, a directed relationship, is indicated by red arrows. Green coloring indicates nodes that have been visited.

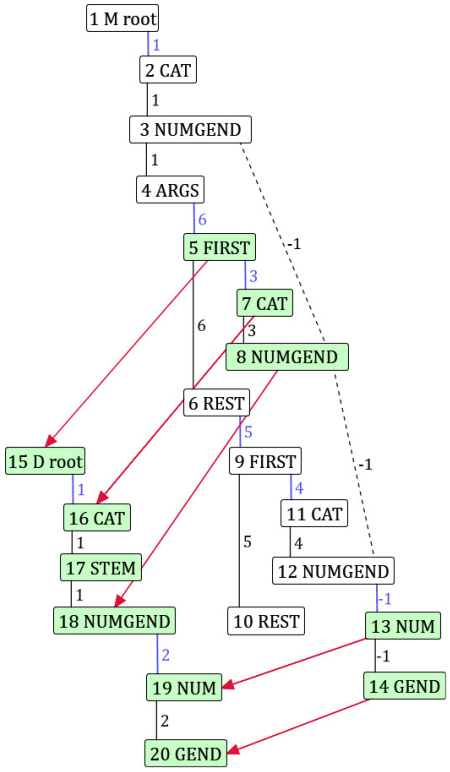
When slot 8 is touched in step 2, its index is associated with the coreferencing slot pointer "-1" in **pp_corefs**. Compare Figure 17 to Figure 19

After step 5, the final step of the first pass, all of the daughter nodes have been visited, but outer structure from the mother remains untouched. This state is examined in more detail in Figure 19.

that the slot is not forwarded.[68] For performance optimization, the t and m_T fields of the slot are cached by copying the *out*-tuple from the source TFS into the unifier slot. This initialization happens on demand, the first time the slot is accessed by the unifier during the operation (as determined by the slot's generation value, which is updated accordingly). The type value stored in the slot represents the current type for the slot, which the unifier can alter if desired. However, the utility doing so is limited, because the unifier cannot alter the set of slots that is selected by querying with the value m_T. Queries using m_T must be submitted to the TFS that owns the slot—since m_T has no meaning otherwise—and that TFS will never provide a valid

[68] The convention of using 0 to terminate forwarding chains is accommodated by the fact that unifier slot zero receives special treatment; it is always excluded from the range of slots assigned to participant TFSes. This can be seen in the illustration, which includes scratch slot zero.

storage index—and by extension, extant unifier slot—for a feature that is not appropriate to the original type value. Because of this, the current implementation only changes a slot's type in one particular case: when type unification between two slots results in a third type which still has no appropriate features.[69]



ps_base, step 5

slot	TFS	m_F	f	gen	ix_{fwd}	t	m_T
0	(global slot 0)			unused			
1	M	(root slot M)		old			
2	M	1	CAT	old			
3	M	1	NUMGEND	old			
4	M	1	ARGS	old			
5	M	6	FIRST		15		
6	M	6	REST	old			
7	M	3	CAT		16		
8	M	3	NUMGEND		18		
9	M	5	FIRST	old			
10	M	5	REST	old			
11	M	4	CAT	old			
12	M	4	NUMGEND	old			
13	M	-1	NUM		19		
14	M	-1	GEND		20		
15	D	(root slot D)			0	pl-word	1
16	D	1	CAT		0	det	0
17	D	1	STEM		0	"these"	0
18	D	1	NUMGEND		0	num-gend	2
19	D	2	NUM		0	pl	0
20	D	2	GEND		0	gender	0

pp_corefs

TFS	coref	slot
M	-1	8

Figure 19. The unifier slots from running example are depicted after the first pass.

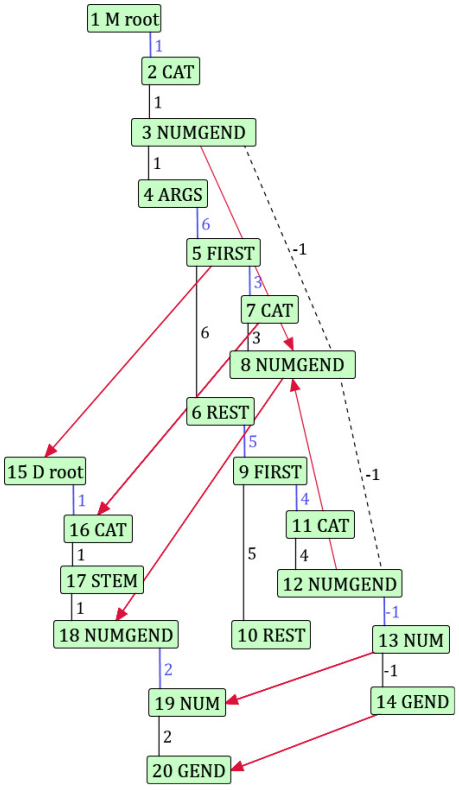
Below the set of slots, the small set of coreference-claiming pointers (described in the introduction to this section) is illustrated. Like **ps_base**, this is a contiguous list of pointers (to scratch slots) that is prepared according to the number of coreferences each participant contains, a figure which is known in advance because it is permanently recorded in each array TFS instance. Further akin to the **ps_base** depiction, the first two columns in **pp_corefs** are for reference purposes only, as this data structure is just an array of pointers whose interpretations can be sufficiently deduced by their list positions (combined with contextual information from unifier control structures). For this example, *M* has one coreference and *D* has none, so only a single uninitialized pointer is present.

For the example case, the unifier joins the two structures in five steps, which are shown in Figure 18. This is essentially the UNION-FIND method described by Aho et al. (1976).

[69] "Still" because the condition entails that the two parent types have no appropriate features. Note also that, in the case described, if the result type *does* have appropriate features, then a well-formedness TFS must be introduced.

The next illustration, Figure 19, shows the state of the slots after the daughter traversal has completed. At this point the success of the overall operation is guaranteed. Several slots are no longer 'old,' but portions of the mother TFS remain untouched, and will be visited for the first time—and copied—while writing the result structure.

As noted at the end of Section 4.4.2, because the array storage model does not physically conflate logically coreferenced nodes, the unifier must take steps to re-establish their referential equivalence for each TFS participant. This is achieved by constructing a representation of the *intra*-participant coreferencing according to the unifier's *inter*-participant forwarding mechanism. Mirroring coreferences from each structure into the unifier is inexpensive and is incrementally completed on demand as nodes are first visited. For example, TFS M has three arcs leading to node $m_T = -1$ (they correspond to unifier slots 3, 8, and 12). One of these was visited during the daughter pass. This is slot 8, as we can see by the fact that the unifier has claimed, on behalf of slot 8, the slot pointer (in **pp_corefs**) dedicated to coreference $m_T = -1$. This ensures that when the writing pass encounters the remaining two slots which have $m_T = -1$ (while completing outer structure M), it will assign the same mark value to all three.



ps_base, step 6

slot	TFS	m_F	f	gen	ix_{fwd}	t	m_T
0	(global slot 0)			unused			
1	M	(root slot M)				phrase	1
2	M	1	CAT			np	0
3	M	1	NUMGEND				
4	M	1	ARGS			cons	6
5	M	6	FIRST		15		
6	M	6	REST			cons	5
7	M	3	CAT		16		
8	M	3	NUMGEND		18		
9	M	5	FIRST			syn	4
10	M	5	REST			null	0
11	M	4	CAT			n	0
12	M	4	NUMGEND				
13	M	-1	NUM		19		
14	M	-1	GEND		20		
15	D	(root slot D)			0	pl-word	1
16	D	1	CAT		0	det	0
17	D	1	STEM		0	"these"	0
18	D	1	NUMGEND		0	num-gend	2
19	D	2	NUM		0	pl	0
20	D	2	GEND		0	gender	0

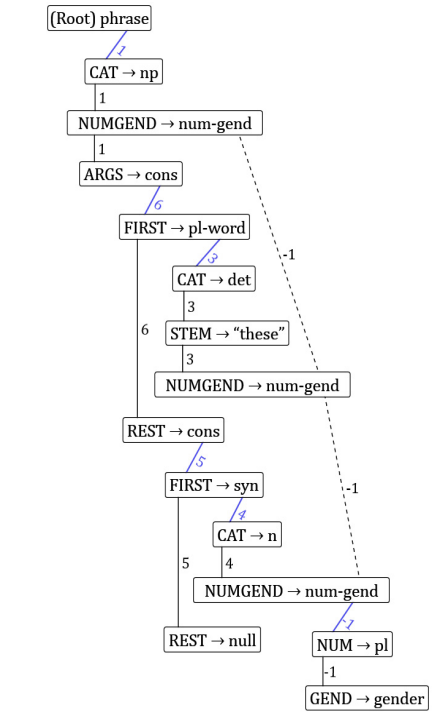
pp_corefs

TFS	coref	slot
M	-1	8

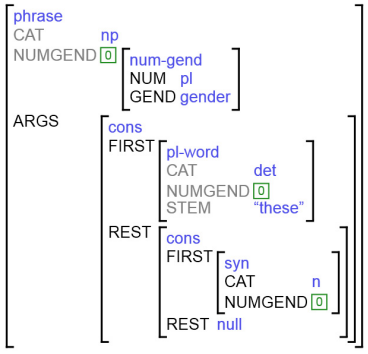
Figure 20. The unifier slots from the example case at the end of the operation, after the writing pass. Notice that the referential equality between coreferenced nodes in ***M***, a property which is absent from the array TFS storage representation, has been restored via the unifier slot forwarding mechanism, as is necessary to ensure the correct conflation of coreferencing.

To illustrate this particular case, consider one of the remaining joined slots, for example, slot 3. The writing pass will encounter this slot for the first time as it copies the untouched mother

structure. Because the slot is 'old,' slot initialization will be triggered, meaning that the out-tuple from the source TFS will be retrieved. Next, noticing that the value of m_T is negative, the unifier checks **pp_corefs** and observes that slot 8 has already been designated for the class. If there were no designation yet, the designation would be claimed.



in-tuple		*out*-tuple	
m_F	f	t	m_T
	ROOT→	phrase	1
1	CAT	np	0
1	NUMGEND	num-gend	-1
1	ARGS	cons	6
6	FIRST	pl-word	3
6	REST	cons	5
5	FIRST	syn	4
5	REST	null	0
4	CAT	n	0
4	NUMGEND	num-gend	-1
3	CAT	det	0
3	STEM	"these"	0
3	NUMGEND	num-gend	-1
-1	NUM	pl	0
-1	GEND	gender	0



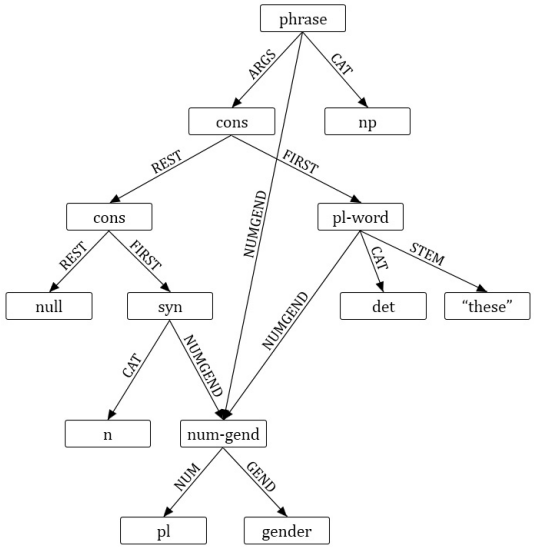


Figure 21. The result structure produced by the unification example, after being written as a new array storage TFS.

At this point, the forwarding value for slot 3 could be set to 8, and this is in fact what happens if slot 8 is not itself forwarded.[70] Recall that a forwarding slot value of 0 indicates the end of a forwarding chain. However, in this case, slot 8 *is* forwarded—across participant boundaries—

[70] In this example, such forwarding is not strictly necessary since each slot is visited only once—and for the last time—during the writing pass, entailing that the slot's forwarding value will never be accessed, and the correct coreferencing in the outer part of the result could, in principle, be determined from information in M. It is during the first pass that re-establishing referential equivalence amongst coreferenced nodes is critical, and generally speaking, the joining pass has no way of predicting the operant condition, namely, how many times a coreferenced node in M will appear within its daughter region.

to slot 18. This is because the unifier chose slot 18 to be the representative slot as it was joining coincident nodes between M and D. Since the unifier must walk this chain to its end anyway, a "free" optimization is possible, namely short-circuiting the chain for slot 3, the newly-visited slot, so that it points directly to slot 18. Thus, in practice the unifier will set 18 as the forwarding value for slot 3, but for clarity the example figures do not assume the optimization.

Prior to beginning the writing pass, the unifier initializes the root slot for the outer TFS, M. If one or more of the other participants were unified into the root node of the designated outer TFS, then this step is unnecessary, since the slot—which signifies the root of the result TFS—would have been initialized during the first pass. Once this slot is initialized, a traversal of the structure embedded in the scratch slots is initiated. This is the result TFS, which is emitted as a set of contiguous 4-tuples as the traversal progresses. The result is a new storage relation, $\mathbb{A}$.

Figure 20 shows the state of the unifier slots after the writing pass. There are no remaining slots belonging to an old generation, which indicates that every slot has now been visited. Type and *out*-mark values for forwarded slots may have been stored, but for clarity they are not displayed here. In practice, type values for slots which are immediately forwarded upon first visit need not be stored. After the writing pass, a new array storage TFS has been produced. It is shown in in Figure 21.

4.6 Summary

This concludes the discussion of Chapter 4, which concerned TFS unification and the array storage unifier. The chapter began with a brief description of TFS unification, and by describing how adopting a well-formedness requirement for typed feature structures implies that any component which produces TFSes—most significantly, a unifier—assumes a responsibility for maintaining the condition.

Section 4.2 presented a literature review of prior work in linguistic unification algorithms. *n*-way unification, a new method investigated as part of this research, was also examined. Although auxiliary to the main thrust of this thesis, the novel method nevertheless serves as a vehicle for raising relevant points of discussion, both theoretical and empirical.

The core of the chapter was the presentation of the array storage unification algorithm. Taken together, this method and the array TFS storage method (presented in Section 2.3 and Chapter 3), comprise the chief contribution of this thesis. The chapter concluded with a diagrammatic walk-through of the operation of the array TFS storage unifier for a simple linguistic example. In the next section, I present an evaluation of the new methods, as implemented in *agree*, a new grammar engineering platform which is wholly based on array TFS storage and its companion unifier.

5 Evaluation

Array TFS storage and the array storage unifier are implemented in *agree*, a new grammar engineering environment which supports the established grammatical formalism and standard practices of the DELPH-IN consortium. *agree* joins the LKB (Copestake 2002b), PET (Callmeier 2002), and Ace (Packard, see footnote 48) as a fourth platform for development and deployment of computational grammars based on the DELPH-IN joint reference formalism (Copestake 2002a), itself a computational variant of Head-Driven Phrase Structure Grammar (HPSG, Pollard and Sag 1994). The typed feature structures comprising these grammars are authored using TDL (Krieger and Schäfer 1994), a declarative text-based notation for describing TFSes and the type hierarchy they extend.

Daughter ARG deletion	Removes daughter substructure after unifying rule daughters
Key driven	Orients active edges according to a grammar condition
Span-only rules	Limits application of tagged rules to spanning structures only
Quick-check	Probes failure-prone paths in the paired argument TFSes prior to unification
Chart dependency filter	Predicates the promotion of each morphology result to the parse phase upon the satisfaction of one or more directed conditions evaluated between the candidate and some other satisfying analysis with a disjoint span.
Ambiguity packing	Controls chart size by allowing a feature structure to be represented by a subsuming surrogate during parsing
Rule-filter	Skips unifications precluded by pre-computed mother-daughter grammar rule compatibilities

Table 5. Common DELPH-IN parser optimizations.

The LKB, PET, and Ace implement Tomabechi's quasi-destructive unification with the structure sharing adaptation described in Malouf et al. (2000), whereas *agree* implements the unifier described in this thesis, and performs structure sharing only in the simple cases described at the end of Section 2.2.2. All four systems offer bottom-up chart parsers and implement a bevy of parsing optimizations and research innovations published by DELPH-IN researchers and others over the past two decades. A selection of these is listed in Table 5; further details are provided in Kiefer et al. (1999) and Oepen et al. (2002).

Because these optimizations are so effective at reducing a parse to just its critical sequence of unifications, it is expected that unifier performance is the most constraining factor in the performance of these parsers. Even so, implemented computational systems involve thousands of free variables which are difficult to control, so end to end evaluation must not be taken as a wholly conclusive measure of the performance of the underlying unification algorithm. For a self-evident demonstration of this, we can see that the LKB and PET, which use the same underlying unification algorithm, present vastly different performance on an identical task. Table 6 shows the time required to parse and exhaustively unpack 287 sentences[71] from the 'Hike'

[71] The full 'hike' corpus contains 330 sentences. For all three systems, 43 sentences which either contain numeric digits (not currently supported by *agree*), or which the LKB could not exhaustively unpack within 2 hours (per sentence) were

corpus, using the English Resource Grammar (Flickinger 2000). To calculate a scaling figure (which expresses the exponent according to which parse time increases with respect to sentence length), each sentence is paired with its parse time and the best least-squares exponential function

$$y = ae^{bx} \tag{5.1}$$

is fit. R^2 is a figure which reflects the accuracy of this type of fit. The derivative of (5.1),

$$\frac{d(\ln a + bx)}{dx} \tag{5.2}$$

(equivalently the slope of a log-linear regression line), is a measure of the degree with which parse time compounds as sentence length (in number of words) increases. Thus linearized, this scaling 'exponent' can be normalized (in this case to *agree*=1.0) for comparison.

	t_{min} (sec.)	t_{max} (sec.)	t_{total} (m:ss)	$y = ae^{bx}$		R^2	scaling (b) *agree*=1.00
				a	b		
LKB	.080	1070.0	67:14	.0133	.4179	.665	2.08
PET	.007	8.4	**1:47**	.0091	.2378	.649	1.18
agree	.008	**7.7**	3:04	.0395	**.2005**	.687	1.00

Table 6. Performance scaling versus sentence length. Bold-faced values show best result. Details on methodology are given in Section 5.4. *agree* concurrency and pipelining is disabled. This summarizes the data of Figure 22, except as noted in Footnote 71.

The two systems which implementations of Tomabechi's unification method exhibit performance scaling that differs by a factor of 1.75. On the other hand, PET and *agree*, which employ quite different storage and unification technologies perform similarly.

Not only do these results confirm the hazards of using system-level parser testing as a proxy for the performance of the unification algorithm, they also disrupt the notion that the managed-*vs.* native-code distinction necessarily delineates classes of system performance. Although *agree* and the LKB are both managed-code systems, in this test their respective results diverge. And as noted above, *agree* and PET, managed and native implementations respectively, show similar performance.

Considering these factors, Table 6 constitutes a sufficient validation of the array storage method and its complimentary unifier. Later in this chapter, Section 5.5 documents a few additional experiments of a more exploratory nature. Before this, however, the next sections provide a technical overview of each evaluated system and document the test methodologies which were observed.

---

omitted, and these are the results shown in Figure 22. The results in Table 6 exclude—for the LKB only—11 additional sentences which took more than 10 minutes for the LKB to complete. This accounts for the 94 minute difference versus Figure 22, but because the time scale is exponential the conclusions remain the same. Additional details on evaluation methodology are provided in Section 5.4.

5.2 *agree*

Development of *agree* progressed over several years in conjunction with the research objectives of this thesis. The system targets the ECMA-335 Common Language Infrastructure (CLI),[72] which is implemented for Windows (".NET") and other platforms ("Mono"). In the interim, *agree* has evolved into a competent DELPH-IN-compliant grammar engineering environment with inherent support for concurrency. In addition to providing a core implementation of the DELPH-IN joint reference formalism, including the set of optimizations listed in Table 5, the system has evolving support for tactical realization of surface strings (sentences) from semantic inputs specified in the format of Minimal Recursion Semantics (MRS, Copestake et al. 2005).

Generation is a recent addition to *agree*. The core functionality of the *agree* generator is in place (including key optimizations such as index accessibility filtering), and further optimizations under development.[73] To support generation, support for trigger rules, surface read-back, skolemization, and MRS extraction and rewriting were added to *agree*. The *agree* parse chart exposes an abstracted edge proximity condition which maximizes the amount of common functionality between parsing and generation. In particular, a novel lock-free mechanism which synchronizes the manipulation of chart edges is shared, allowing both modes to deploy proactive and retroactive ambiguity packing (Oepen and Carroll 2000b)—in a new concurrent adaptation. This work will be described in a separate publication.

The core functions in *agree* are implemented in a library module such that they are only accessible through programmatic interfaces. Accordingly, several client applications which control and provide access to the system have been developed. The most basic is a console process which loads a grammar, and parses (or generates) one or more input items. Input adapter modules allow inputs to be supplied from different types of sources, such as from the console, from an `[incr tsdb()]` profile (Oepen and Carroll 2000a) or from a simple text file. This console program is the control harness that was used to evaluate *agree* in this thesis.

The core library and console application are compatible with, and have been tested on, both of the target environments, .NET and Mono. In addition to the console front-end, *agree* also sports a few different graphical user interfaces[74] which variously support visualization and interactive manipulation of grammar entities, and specialized grammar diagnostics. Future plans include a program which enables a rich editing workflow for grammar engineers. Discussing these programs is beyond the scope of this thesis.

---

[72] European Computer Manufacturer's Assocation, see footnote 27 on page 4.

[73] Certain parts of the generator have been developed with the assistance of Spencer Rarrick.

[74] These make use of WPF, a vast, next-generation system of .NET system libraries that, sadly, will not be available on the open-source Mono platform. The dependency extends to the *agree* library that provides graphical TFS, MRS, parse tree, and parse chart rendering. Aware of this penalty, the route was chosen because the rapid GUI development that WPF enables was too compelling.

Facilitated by *agree's* ability to read configuration settings in either LKB or PET format, several grammars are regularly tested. These include the English Resource Grammar (Flickinger 2000), the JACY Japanese grammar (Siegel and Bender, 2002), a grammar of Mandarin Chinese (Zhang 2011), and my own grammar of the Thai language.[75] The latter two grammars are based on the Grammar Matrix (Bender et al., 2002), which includes a customization system (Bender et al., 2010) that facilitates the rapid prototyping of new grammars for typologically diverse languages.

The DELPH-IN consortium is committed to open-source development. The source code for *agree* is released under the MIT license and will be available for community comment and collaborative development. With the exception of the graphical client programs mentioned above (because they require WPF) *agree* can be built and tested with Mono, an open-source environment which supports all modern hardware platforms, including Windows, Mac, and Linux.

5.3 Existing systems

The LKB, evolved from earlier systems which date to the early 1990s, supports both parsing and generation, and—with its graphical user interface—facilitates interactive grammar engineering. Ace is a speedy native-execution parser developed by Woodley Packard which implements support for advanced features such as generation and selective unpacking; formally released in 2011, this system is not included in this evaluation. PET, originally developed as a platform for experimentation in parsing and unification technologies, is now the first-choice for batch-oriented parsing in production and high-volume research environments; it contains state-of-the-art statistical parse selection and *n*-best selective unpacking. PET enjoys continued, active development; in its distributed release—or as a source-code branch—it is widely used for grammar development, parser research, and grammar instrumentation.

Because the PET system prepares a compiled image of a grammar as a first step, it can also facilitate experimental parser development. An example is the work mentioned in Section 4.2.10; by designing his thread-safe parser CaLi so that it loaded PET grammar images, van Lohuizen (2001, 96) did not have to implement the infrastructure for parsing TDL files, closing the type lattice, and other tasks associated with loading DELPH-IN grammars.

Existing DELPH-IN parsers are single-threaded.[76] For several reasons, there has historically been little reason to develop concurrent systems within the consortium's research program. First, questions of grammar engineering are equivalently answered between single-and multi-threaded parsers. Second, multi-core systems—and decline in the progression of CPU core frequency increases—have only taken hold in recent years. Third, parsing and generation are

[75] The Thai grammar was initially developed as part of coursework supervised by Emily Bender. The grammar incorporates some linguistic analyses published by Nuttanart Facundes, who also offered language insights for the project during my lectureship at King Mongkut's University in Thonburi, Thailand.

[76] The CaLi system developed by Marcel van Lohuizen appears to be no longer maintained, so it is not considered a current DELPH-IN system.

thought to be “embarrassingly parallel” tasks; if the overall task requires processing more than one sentence, multiple CPUs might be trivially harnessed by spooling sentences to several single-threaded processes.

Subtended by the third consideration, however, are two assumptions that may not hold: that the processing is batch-oriented, and that provisioning multiple parser instances is functionally equivalent to finer-grained concurrency. Clearly the first assumption is not applicable to some applications. The performance objective in real-time applications, for example, is simply to minimize, by any means possible, the parse time for a single sentence. Text stream sources which establish a real-time requirement include voice recognition systems and web interactions. As for the second assumption, two counterexamples are summarized in Section 5.6.

5.4 Methodology

For consistency, the test configuration used swappable hard drives to alternate between Linux (LKB/PET) and Windows (*agree*) on the same machine; it is an 8-core (2 × X5460) with 32GB of RAM. The Mono build of *agree* was not performance tested.

In accordance with the research concerns of this thesis, a stress configuration is used for all testing in order to maximally expose the efficiency of each parser’s unifier and its underlying TFS storage model. Specifically, the parsers are always configured for exhaustive unpacking of every derivation in the parse forest, for every sentence. In the case of PET, the `[incr tsdb()]` (Oepen and Carroll 2000a) profiling system was configured such that PET’s ‘cheap’ parser operated with the following options:

```
packing=7 -default-les -cm english.grm
```
(5.3)

Because the `-nsolutions` option and packing bit `0x8` are not specified, PET will unpack the entire parse forest. Unfortunately, the LKB is not able to complete exhaustive unpacking for several of the sentences, owing to resource exhaustion. Sentences for which the LKB consumes all 32GB of available physical memory *and also* has not completed the individual sentence after two hours are abandoned and notated as having not finished. Finally, for all tests, and for all systems, timings do not include grammar start-up.

5.4.2 Evaluation grammar and corpus

The evaluation grammar is the English Resource Grammar (ERG, Flickinger 2000) revision 10342. This is a publically available broad-coverage precision grammar, the most comprehensive of its kind. The grammar is distributed with the LinGO Redwoods Treebanks (Oepen et al. 2002), a set of curated corpora with reference parses. Tests were conducted with the ‘hike’ set, a collection of 330 sentences gleaned from Norwegian tourism brochures which is often used in DELPH-IN parser evaluation. The average sentence length is 11.67 words, and every sentence is fully covered in the lexicon. The task entails performing around 20 million top-level unifications.

As noted in Section 5.4, the LKB is not able to complete exhaustive unpacking of a number of sentences in the 'hike' corpus. In an attempt to address this problem early in this thesis research, these complex sentences were excluded from the test corpus, yielding a subset which contains 287 of the 330 'hike' sentences (the 43 excluded sentences also includes 12 sentences which contain numeric digits, since they are not currently accepted by *agree*). Because a large number of evaluation trials was archived on this basis, and in order to support consistent comparison with these data, this is the corpus that is used in the batch processing evaluation, presented in Section 5.5.1. Note however, that this advisory applies neither to the real-time requirements evaluation (Section 5.5.2) nor the throughput analysis (Section 5.5.3), which both use the 318 'hike' sentences which do not contain numerals.

5.4.3 Correctness

Correctness of the *agree* results was established over the 'hike' corpus. On the assumption that LKB and PET were cross-validated in previous work, derivation correctness was only validated in full between the *agree* and PET results. Discounting well-understood differences[77] which account for around 0.6% of the 277,946 derivations, derivation correctness was exact.

This validation was performed by rendering text-based (path list) feature structures using the PET feature ordering, filtering out token-mapping paths (*agree* currently does not inject standoff positions into the feature structure), and comparing with a text 'diff' utility. The entire process was automated. In contrast to the performance tests, *agree* derivation correctness was also established for the Mono/Linux. In this test, full results from *agree* were compared between its Windows and Linux builds, and they were found to be identical.

5.5 Results and analysis

This section presents results from a variety of experiments which aim to characterize the performance of *agree*'s managed-code parser against two existing parsers, one managed and one native, as a baseline. Parser comparison is notoriously difficult (Dridan 2010); results from this section should be calibrated accordingly. Dridan suggests organizing evaluation around high-level application scenarios. Two basic scenarios are contrasted. In the first, batch processing, the objective is to parse, as quickly as possible, a set of sentences which are all available at the outset. Within this scenario, *agree* is evaluated with concurrency disabled. Naïve concurrency, as it applies to existing parsers, is primarily discussed in Section 5.5.3, but is not evaluated in the two batch-oriented test configurations of Section 5.5.1, for reasons that will be noted in the text.

The second test scenario concerns applications which express a real-time requirement (Section 5.5.2). In this case the objective is to parse a single sentence as quickly—and by whatever means—possible. Modes which allow existing parsers to compete in this category are briefly

[77] A single difference in tokenization can result in a large difference in the number of derivations. Twelve differences in tokenization accounted for 1,668 derivations not generated by one system or the other. Each difference was carefully investigated so that derivations that were not affected by the difference were correctly cross-validated.

surveyed. Finally, in Section 5.5.3, experiments exploring *agree's* throughput and scaling are discussed, with analysis.

5.5.1 Batch parsing

This section examines batch parsing performance, with experimental contrasts between the LKB, PET and *agree* parsers. Two setups are tested. First, since any parser can be configured for sentence pipelining—that is, accepting sentences from a central spooler—*agree* is tested with all concurrency disabled. To understand the rationale in this test design requires a summary understanding of *agree's* concurrency modes. A brief overview is provided in the following paragraphs.

agree supports two distinct concurrency mechanisms. First, a sentence spooler is a built-in capability it explicitly exposes. Zero or more *submitter* instances can be created to spool sentences to a corresponding parser instance. Each operates within the same process and on the same grammar, without requiring the expense of an additional OS process or the memory cost of redundantly loading the grammar.

The second concurrency mechanism in *agree* is a configuration option, exposed by the parser, which places a limit on the number of tasks which it may submit to the Common Language Runtime (CLR). This mechanism is independent of the number of submitters provisioned. To understand the effect of this limit, a brief review of the parser's scheduling is discussed next.

Fundamentally, the *agree* parser issues fire-and-forget tasks to the CLR's lightweight task dispatcher, `Task<T>`,[78] on a linguistic basis—that is, according to conditions specific to the grammar and the sentence being parsed. Each parse task conceived by *agree* may invoke up to several dozen unifications before exiting. For example, one task is dedicated to evaluating a new passive chart edge against all pre-existing active edges, so the number of unifications depends on the sentence length and complexity. Another task generates new active edges for the new passive chart edge, so the amount of work performed by this task depends on the number of grammar rules.

If dispatching a new task would exceed the task limit, then the work is immediately executed by the executing thread (the caller who attempted to queue the task). Otherwise the task is queued to the operating system and *agree* has no further involvement in the scheduling or prioritizing of the task, except to decrement the total task count when the task eventually completes. Thus, the 'parser task limit' mechanism places an upper limit on the number of tasks that the CLR's `Task<T>` subsystem becomes aware of. Because `Task<T>` is well designed, this

[78] `Task<T>` is responsible for mapping application-submitted tasks to a limited set of threads from the system thread pool, and for dynamically monitoring the number of threads and other parameters so as to maximize performance. It is a general-purpose concurrency service which has no knowledge of *agree's* linguistic parsing application. The `Task<T>` facility is highly regarded for its sophisticated self-tuning capabilities and for incorporating results from contemporary concurrency research, including runtime tuning via empirical hill-climbing and work-stealing.

limitation rarely improves performance, and it is largely used for performance testing, as in the experiments presented here.

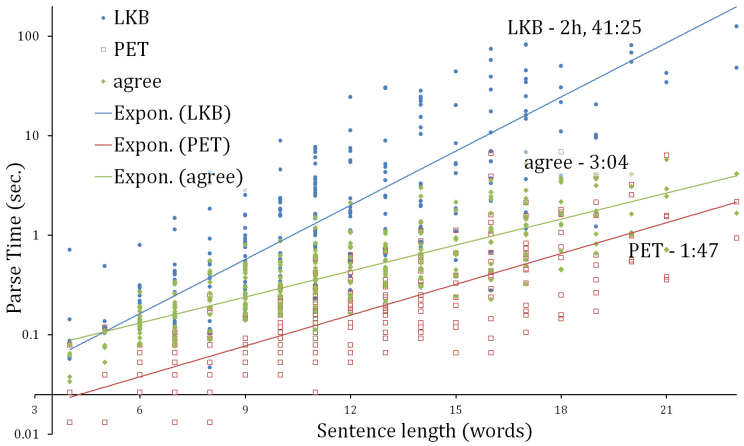


Figure 22. Batch processing performance on the 'hike' corpus. Discussion can be found in the chapter introduction.

Beyond the independent configurations of the sentence submitter and the parser task limit, there is no single-threaded mode in *agree*. Instead, the single-threaded restriction in *agree* amounts to provisioning a single submitter and setting the parser task limit value to 1. Because one task is always assumed to be running, doing so effectively forbids the chart parser from creating new threads for agenda tasks.

Single-threaded operation could be construed as a common ground between the systems. The rationale for limiting *agree* to single-threaded mode is the assumption that throughput scaling—whether achieved through multiple parser processes or through intrinsic concurrency—would benefit all systems relatively equally. Unfortunately, as the preceding discussion has intimated, because *agree* does not differentiate a tasking limit of 1 from any other value, its concurrency expenses—such as the use of atomic processor instructions and, significantly, the act of forming closures over parser work items—are still active when it is limiting itself to a single thread. Single-threaded batch processing performance is shown in Figure 22. These results were discussed in the introduction to this chapter.

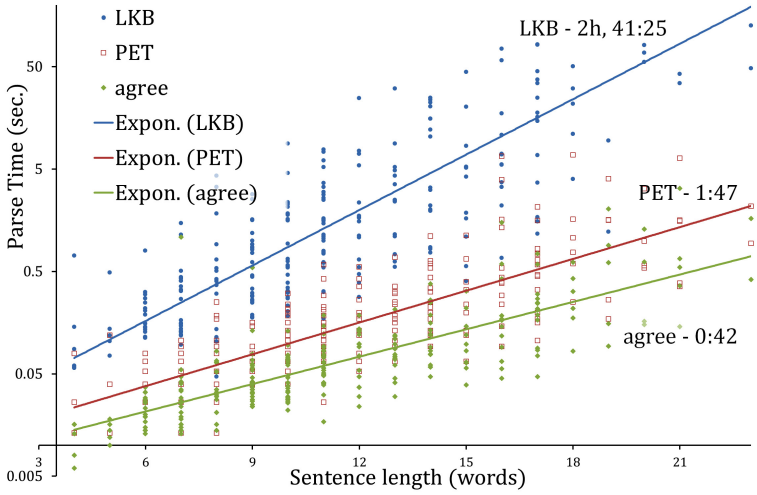


Figure 23. Single-sentence real-time performance on the 'hike' corpus.

The second batch processing model admits that penalizing *agree* by saddling it with concurrency overheads that go unused may be just as arbitrary as comparing single- to multi-threaded results. In this test, the corpus is exhaustively parsed and unpacked by each parser according its

best means. With 8-way concurrency enabled, *agree* parses the corpus in 42 seconds (Figure 23).

The introduction to this chapter stressed the difficulty of interpreting end-to-end parser testing results, and presented data to illustrate the point. With this caveat in mind, the result of this experiment (Figure 22) can be described in general terms. Despite being configured with concurrency disabled, *agree* shows the best scaling performance, meaning that it exhibits the smallest increases in slowdown as sentences get longer. However, the sentences in the test corpus are not long enough, on average, for this factor to guarantee the fastest parsing for the corpus as a whole, and PET completes the task fastest.

5.5.2 Real-time requirement

The intention of the second overall evaluation scenario is to model applications with real-time requirements, such as speech processing or web interaction. Here, the objective is to obtain the fastest result for a single sentence. Sentence pipelining is not relevant because there is only a single sentence to process. As with the second batch processing test, any attributes particular to a parser which do not alter its linguistic results are permitted. The discussion of this section naturally overlaps with the discussion of parsing throughput which follows in Section 5.5.3.

> Today we might easily get tempted to take a hike on the glacier when we look down on the glacier from the Oslo plane. (5.4)

For these tests, exhaustive parsing and unpacking of sentence (5.4) (giving 195,605 analyses) is studied. This is one of the sentences that was excluded from the tests performed in Section 5.5.1 because it could not be exhaustively unpacked by the LKB. The same sentence will be used to analyze the throughput scaling of *agree* and PET in Section 5.5.3. The job requires a large number of top-level unifications: about 500 in morphological analysis, 100,000 in the main parse, and 900,000 in exhaustive unpacking. To support that discussion, the presentation begins with detailed results from *agree*. Comparison with the baseline parsers will be presented shortly.

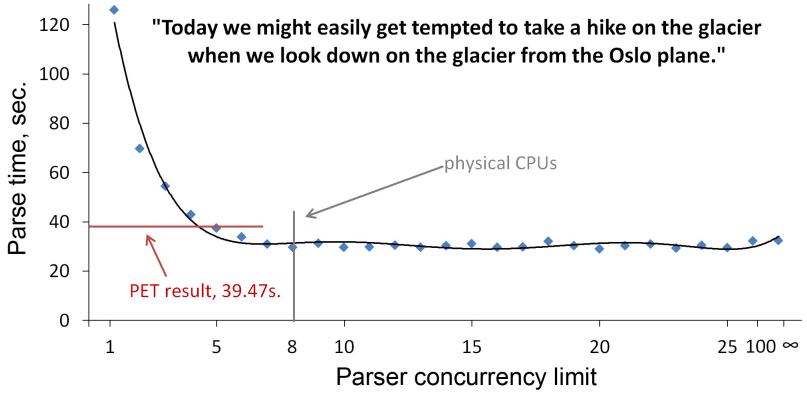


Figure 24. 8-CPU performance scaling of the *agree* parser with a 24-word sentence. When the parser would exceed the task limit, it executes the closure immediately instead of queuing another task to the runtime. Performance improvement is not yet constrained by other resource limitations when adding the eighth physical processor (the maximum configuration tested). Beyond this point, no over-commitment penalty is seen.

Sentence (5.4) is parsed and exhaustively unpacked with the *agree* parser concurrency limited over the range $\{1, 2, \ldots 25, 100, \infty\}$. Configuring the parser with an

infinite task limit means that every task conceived by the parser is queued, which delegates full responsibility for managing processor over-commitment to `Task<T>`. Results, shown in Figure 24, show competent scaling for *agree*. After exhibiting poor performance of 126 seconds when limited to just one processor, parse time monotonically decreases as additional processors are permitted. As expected on a machine with eight physical CPUs, a near-minimum value of 28.19 seconds occurs when the parser limits itself to creating eight tasks. This is a bit surprising because not all phases of the parse task are expected to be able to submit that many tasks. For example, after morphological analysis but before the main parse, chart dependencies are evaluated as a single-threaded operation. The explanation is that, as intended, this test is overwhelmingly dominated by unification work.

System		**parse time, sec.**
LKB		d.n.f., > 7200.00
PET		39.47
agree	parser task limit = 1	126.13
	min (at parser task limit = 8)	28.19
	parser task limit > 8 (avg.)	29.84

Table 7. Comparison of the results from Figure 24 with other parsers.

When physical processors are over-committed, the `Task<T>` system exhibits sensible behavior. Average parse time in this region is 29.84 seconds with a standard deviation of only 1.13 seconds (3.8%). Table 7 extends the analysis of Figure 24 to the LKB and PET. With only a single processor, PET parses the sentence in 39.47 seconds, clearly surpassing both the LKB (which was not able to complete the task within two hours), and *agree*, which took just over two minutes.

5.5.3 *agree* parser throughput and scaling

To evaluate the benefit of concurrency to the *agree* implementation, scaling was studied across a range of concurrency levels. The objective was to determine the degree to which contention degrades parsing performance. Concurrency overheads can be categorized as those for which *agree* is responsible, and those which occur within the runtime environment's `Task<T>` facility. In the former category, primarily, is contention in the lock-free parse chart. Details of this component are beyond the scope of this thesis, but in short, each cell of the parse chart is independently protected by an extremely brief "optimistic" sequence of atomic processor instructions. This means that every chart operation always assumes that it will succeed, and proceeds without locking, protected only by detecting rejected CMPEXCH results (expected value collisions). When detected, this contention requires that the operation be retried anew. The mechanism is simple, but suffers extreme penalties when multiple CPU cores crowd around a popular chart cell.

Results from the real-time requirements testing were used to analyze whether contention was negatively affecting *agree*'s CPU scaling. If it were, parse time might begin to trend *upwards* before reaching the number of physical processors. This is not observed, so contention does

not seem to be implicated. This same data could also provide evidence of disproportionate administrative overheads associated with the CLR's tasking support. If drastically overcommitting the runtime's task pool incurred a steep penalty, parse time for the unconstrained configuration—which actually ends up queuing a peak of about over 4,313 parser tasks for this one sentence—would rise in contrast to the parse time for, say, 5 CPUs. This, too, is not observed, suggesting that `Task<T>` is, as claimed, sophisticated and well-tuned.

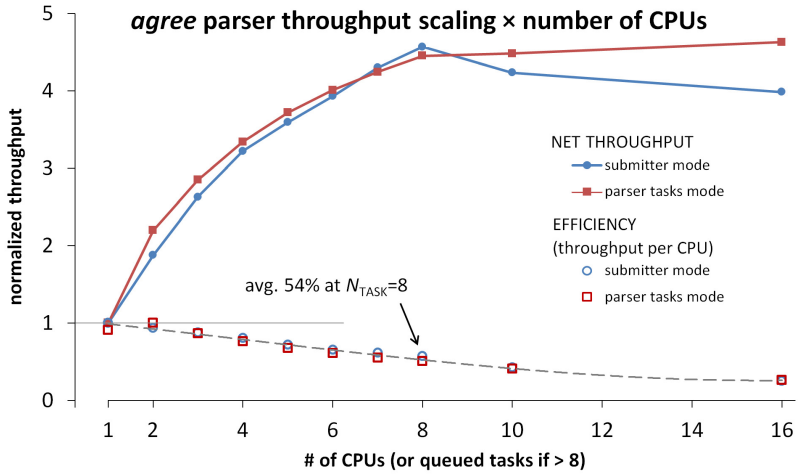


Figure 25. Comparison of *agree's* scaling mechanisms—the parser task limit, and the number of submitter instances—aggregated over the 'hike' corpus. The *y*-axis is normalized to the single-task performance for either method.[79]

Beyond the overheads internal to `Task<T>` just mentioned above are overheads associated with creating closures over the parser work items, in order that they can be submitted to `Task<T>`. In the current design, this penalty is borne even when the parser is configured for single-threaded operation. In fact, the expensive atomic processor instructions that protect the lock-free chart, and several other concurrency considerations that become unnecessary, are all still observed when only a single CPU is active. As mentioned in section 5.5.1, this complicates evaluation because it entails that limiting *agree* to single-threaded operation does not trivially enable conclusive performance comparison between its unifier and storage systems and those of those of the single-threaded control group.

In Figure 25, the two distinct scaling mechanisms available in *agree* are compared. The performance data are aggregated over parsing the entire 'hike' corpus and independently normalized, for each mechanism, to 1.0. In the parser tasks method, the parser limits the number of tasks it creates, as described in Section 5.5.1. This method is compared to the submitter method. Though sharing the same grammar, each submitter supervises the submission of one sentence at a time to a parser instance. Sentences can be spooled from a source shared amongst multiple submitters, or from entirely separate sources. Thus the second tested configuration is the number of submitters provisioned, where each submitter's parser instance is configured to limit itself to creating a single task.

[79] Thanks to Woodley Packard for suggesting this presentation format.

These data indicate good scaling throughout the range of physical CPUs. The submitter mechanism is more coarse-grained than the parser tasks mechanism; as expected, it incurs greater degradation when the number of CPUs is oversubscribed (greater than eight). Oversubscription of parser tasks fares better, a credit to the operating system facility which adaptively manages their scheduling.

Dividing each net throughput observation by the total number of CPUs it engages gives throughput per CPU. The values are also normalized—independently for each mechanism—to 1.0, and linear trends are then plotted. A perfectly horizontal line is optimal, since it would indicate that adding additional processors has no effect on those already operating. With eight processors, overall performance has deteriorated to 57% of what would be expected if there were no concurrency overhead for the submitter, and to 51% for the finer-grained mechanism.

The data highlight a few positive results. First and most obviously, performance losses from medium-grained concurrency are nearly identical to those from the embarrassingly parallel approach. Reviewing what each is expected to be susceptible to, both will experience contention for system memory bandwidth, but only the former should also experience shared data contention and cache miss penalties.[80] That they decrease in such close accord is an important result for this thesis and for *agree* as a whole, because it means that its ability to bring multiple processors to bear on a single parsing task—that is, on parsing a single, complex sentence—does not imply disproportionate waste of the machine's resources. Across the entire range tested in Figure 25, the more flexible, finer-grained mechanism adds the capability of accommodating real-time parsing requirements with no sacrifice of *agree's* fundamental unification throughput.

The preceding discussion suggests that, for batch processing, there is little need to worry about which *agree* concurrency mode or modes are used, and experiments largely bear this out. Nevertheless, miniscule—but repeatable—performance differences are obtained when provisioning different submitter and parser task limit configurations. In fact, peak performance on the 8-core test machine is obtained when using two submitters, each limited to queuing a dozen tasks or so. This balance between coarse-grained and finer-grained tasking maintains a 2-3% performance improvement versus other configurations. An intuitive explanation for the result is that having more than one task running at all times helps smooth over the brief phases of single-threaded activity (tokenization, chart dependencies, etc.) inherent in parsing a sentence; but more than two coarse-grained tasks causes memory strain because the simultaneous parsing of unrelated sentences produces too many intermediate analysis structures.

[80] Even though multiple submitters share the same grammar, their access is read-only, which should cause no contention. The exception is the case of *false sharing*—described in footnote 57—which is easy to fix, but hard to detect and locate.

A second notable result from Figure 25 is that, with the finer-grained mechanism, increasing from one CPU to two increases the throughput per CPU slightly, reaching the normalization maximum with two—rather than just one—active processor. The result indicates the degree to which contention affects the parser in single-threaded mode. Concurrency overheads—the so-called 'communication' penalties—inherent in having multiple threads work on the same task, are found to be minimal. **Figure 26** zooms in to the interesting part of the graph.

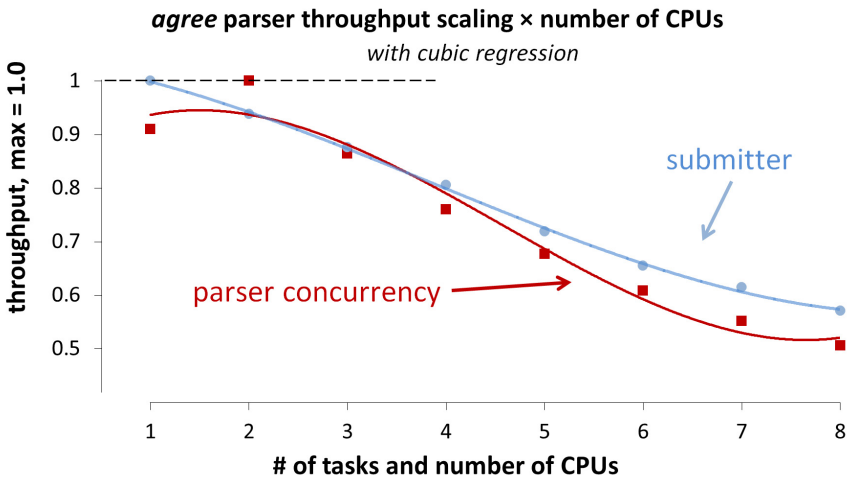


Figure 26. Detail from Figure 25. The overheads from agree's medium-grained concurrency are exposed.

Although the slight degree of concurrency overhead is properly attributed to the parser component—which is not itself a subject of this thesis—it is important confirm that its implementation details are not significantly coloring *agree's* scaling and throughput results.

5.6 Evaluation summary

This chapter evaluated array storage and the array storage unifier in realistic grammar processing tasks by exercising them with *agree's* concurrent chart parser. Overall, the new methods are deemed competitive contributions to contemporary research. Concurrency support adds some overhead to *agree* which cannot be disabled, so the system becomes most competitive when running on multi-core hardware. This property is most relevant to applications with real-time requirements, where parse results for complex sentences are needed as quickly as possible, and without regard for computational expense. In this test, concurrency allows the new system to parse and exhaustively unpack a twenty-four word sentence 24% faster than the comparison system.

In *agree*, two distinct concurrency features—sentence spooling and finer-grained parser tasking—are both coordinated by the same adaptive supervisor. This flexible model simplifies configuration and provisioning, enabling *agree* to rapidly and automatically respond to changing parsing conditions. This suggests a scenario for which *agree* is ideally suited: the batch processing of corpora which contain unpredictable variation in sentence complexity. Because *agree* dynamically adapts its parser concurrency level based on a process-wide consideration of memory consumption, it is able to add more tasks when processing simple inputs, and hold back on starting additional sentences when working on a complex input.

6 Conclusion

This thesis describes a new approach to the storage of typed feature structures appropriate for linguistic modeling with constraint-based unification grammars. Nodes comprising a single TFS are stored as tuples in a fixed-size, read-only array which serves to consolidate a large number of system allocations into just one.

Also described is a unification algorithm suited to the new storage paradigm. By incorporating reference to the mixed-arity feature-value maps of the participating input structures in its description of the putative result structure, the unification algorithm obtains a simplifying guarantee, namely, that the result structure is necessarily latent within that joint set of maps. The formal result is that the unifier requires no operationally variant data structures relevant to the traversal conditions, and is always able to sufficiently express the output TFS within an exclusively *a priori* framework.

The techniques are demonstrated in *agree*, a new DELPH-IN compatible grammar engineering environment which was developed for this thesis. Comparisons between *agree* and existing parsers show that array TFS storage and its companion unifier are effective. Evaluated on an identical task, the new system manifests performance in league with a widely-used high-performance system. For comparison purposes, these tests required that *agree's* concurrency features be disabled, but in other testing, the system's concurrency scaling has proven to be one of its most compelling qualities.

7 References

Hassan Aït-Kaci. 1984. A lattice theoretic approach to computation based on a calculus of partially ordered type structures. Ph.D. Thesis, University of Pennsylvania, Philadelphia, PA.

Hassan Aït-Kaci, Robert Boyer, Patrick Lincoln, and Roger Nasr. 1989. Efficient Implementation of Lattice Operations. *ACM Transactions on Programming Languages and Systems*, 115-146.

Emily M. Bender, Dan Flickinger, and Stephan Oepen. 2002. The grammar matrix: an open-source starter-kit for the rapid development of cross-linguistically consistent broad-coverage precision grammars. in John Carroll et al., eds, *Proceedings of the workshop on grammar engineering and evaluation (COLING-GEE 2002), vol. 15*. Association for Computational Linguistics, Stroudsburg, Pennsylvania, 1-7.

Emily M. Bender, Scott Drellishak, Antske Fokkens, Laurie Poulson, and Safiyyah Saleem. 2010. Grammar Customization. *Research on Language & Computation 8(1)*. Springer Netherlands. 23-72.

Ulrich Callmeier. 2000. PET: a platform for experimentation with efficient HPSG processing techniques. *Natural Language Engineering* 6(1): 99-107.

Ulrich Callmeier. 2001. *Efficient Parsing with Large-Scale Unification Grammars*. MA Thesis, Universität des Saarlandes—Fachrichtung Informatik.

John Carroll. 1993. *Practical Unification-based Parsing of Natural Language*. Ph.D. Thesis. University of Cambridge.

Edgar Frank Codd. 1970. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6), 377-387.

Alain Colmerauer, 1982. *Prolog II Reference Manual and Theoretical Model*. Internal Report, Groupe d'Intelligence Artificielle, Faculte des Sciences de Luminy, Marseilles, Universite d'Aix-Marseille.

Ann Copestake, 1993. *The Compleat LKB*. University of Cambridge Computer Laboratory, Technical report No. 316 and ACQUILEX-II Deliverable, 3.1

Ann Copestake. 2002b. Definitions of Typed Feature Structures. in Oepen et al., eds., *Collaborative Language Engineering*. CSLI Publications, Stanford California, 227-230.

Ann Copestake. 2002a. *Implementing typed feature structure grammars*. CSLI Publications, Stanford California.

Ann Copestake and Dan Flickinger. 2000. An open-source grammar development environment and broad-coverage English grammar using HPSG. in *Proceedings of the Second conference on Language Resources and Evaluation (LREC-2000), Athens, Greece*.

Ann Copestake, Dan Flickinger, Carl Pollard, and Ivan A. Sag. 2005. Minimal recursion semantics: An introduction. *Research on Language & Computation*, 3(4):281–332.

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

Rebecca Dridan. 2010. *Using lexical statistics to improve HPSG parsing*. Ph.D. dissertation, Universität des Saarlandes.

Martin C. Emele. 1991. Unification with lazy non-redundant copying. in *Proceedings of the 29th annual meeting on Association for Computational Linguistics (ACL 1991)*. Association for Computational Linguistics, Stroudsburg, Pennsylvania, 323-330.

Dan Flickinger. 2002. On building a more efficient grammar by exploiting types. *Natural Language Engineering*, 6, 15-28.

Gerald Gazdar, Ewan Klein, Geoffrey K. Pullum, and Ivan Sag. 1985. *Generalized phrase structure grammar*. Harvard University Press, Cambridge Massachusetts.

Kurt Godden. 1990. Lazy unification. in *Proceedings of the 28th annual meeting on Association for Computational Linguistics (ACL 1990)*. Association for Computational Linguistics, Stroudsburg, Pennsylvania, 180-187.

Herbrand, Jacques. 1930. Recherches sur la theorie de la demonstration. Ph.D. Thesis. In W. Goldfarb, ed., 1971. *Logical Writings of Jacques Herband*. Harvard University Press, Cambridge Massachusetts.

Mark Johnson. 1987. *A Logic of Attribute-Value Structures and the Theory of Grammar*, Ph.D. Thesis, Stanford University, Stanford, California.

Lauri Karttunen. 1986. D-PATR: a development environment for unification-based grammars. in *Proceedings of the 11th conference on Computational linguistics (COLING 1986)*. Association for Computational Linguistics, Stroudsburg, Pennsylvania, 74-80.

Bernd Kiefer, Hans-Ulrich Krieger, John Carroll, and Rob Malouf. 1999. A Bag of Useful Techniques for Efficient and robust Parsing. in *Proceedings of the 37th annual meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, Stroudsburg, Pennsylvania, 473-480.

Hans-Ulrich Krieger and Ulrich Schäfer. 1994. TDL: a type description language for constraint-based grammars. in *Proceedings of the 15th conference on Computational linguistics (COLING 1994), Vol. 2*. Association for Computational Linguistics, Stroudsburg, Pennsylvania, 893-899.

Stephan Oepen and John Carroll. 2000a. Parser engineering and performance profiling. *Natural Language Engingeering* 6, 1 (March 2000), 81-97.

Stephan Oepen and John Carroll. 2000b. Ambiguity packing in constraint-based parsing: practical results. in *Proceedings of the 1st North American chapter of the Association for Computational Linguistics conference (NAACL 2000)*. Association for Computational Linguistics, Stroudsburg, Pennslyvania, 162-169.

Stephan Oepen, Kristina Toutanova, Stuart Shieber, Christopher Manning, Dan Flickinger, and Thorsten Brants. 2002. The LinGO Redwoods treebank motivation and preliminary applications. in *Proceedings of the 19th international conference on Computational lin-*

guistics (COLING 2002) Vol. 2. Association for Computational Linguistics, Stroudsburg, Pennslyvania, 1-5.

King, Paul. 1989. *A Logical Formalism for Head-Driven Phrase Structure Grammar*. Ph.D. Thesis, University of Manchester.

King, Paul. 1994. *An expanded logical formalism for Head-Driven Phrase Structure Grammar*. Arbeitspapiere des SFB 340, Bericht Nr. 59, University of Tübingen.

Knight, Kevin. 1989. Unification: A Multidisciplinary Survey. ACM Computing Surveys 21(1), 93-124.

Kiyoshi Kogure. 1990. Strategic lazy incremental copy graph unification. in Hans Karlgren, ed., *Proceedings of the 13th conference on Computational linguistics (COLING 1990) vol. 2*. Association for Computational Linguistics, Stroudsburg, Pennsylvania, 223-228.

Robert Malouf, John Carroll, and Ann Copestake. 2000. Efficient feature structure operations without compilation. *Natural Language Engineering,* 1(1):1-18.

Kurt Mehlhorn. 1984. *Data structures and Algorithms 2: Graph Algorithms and NP Completeness*, *Volume 2*. Springer-Verlag, Heidelberg.

Stephan Oepen and John Carroll. 2000. Ambiguity packing in constraint-based parsing: practical results. in *Proceedings of the 1st North American chapter of the Association for Computational Linguistics*. Morgan Kaufmann Publishers Inc., San Francisco, California, 162-169.

Stephan Oepen, Kristina Toutanova, Stuart Shieber, Christopher Manning, Dan Flickinger, and Thorsten Brants. 2002. The LinGO Redwoods Treebank: Motivation and preliminary applications. in *Proceedings of the 19th International Conference on Computational Linguistics (COLING 2002), Taipei, Taiwan*. Association for Computational Linguistics, Stroudsburg, Pennsylvania.

Stephan Oepen, Dan Flickinger, Jun-ichi Tsujii, and Hans Uszkoreit, eds. 2002. *Collaborative Language Engineering: A Case Study in Efficient Grammar-Based Processing*. CSLI lecture notes, CSLI Publications, Stanford California.

Fernando C. N. Pereira. 1985. A structure-sharing representation for unification-based grammar formalisms. in *Proceedings of the 23rd Annual Meeting of the Association for Computational Linguistics. Chicago, IL, 8-12 July 1985*, Association for Computational Linguistics, Stroudsburg, Pennsylvania, 137-144.

Fernando C. N. Pereira and D Warren. 1986. Definite clause grammars for language analysis. in Barbara J. Grosz, Karen Sparck-Jones, and Bonnie Lynn Webber, eds., *Readings in natural language processing*. Morgan Kaufmann Publishers Inc., San Francisco, California, 101-124.

Carl Pollard and Ivan A. Sag. 1994. *Head-Driven Phrase Structure Grammar*. University of Chicago Press and CSLI Publications, Stanford, California.

Stuart M. Shieber, Lauri Karttunen, and Fernando C. N. Pereira. 1984. *Notes From The Unification Underground: A Compilation Of Papers On Unification-Based Grammar Formalisms*. Technical Note 327. AI Center, SRI International

Stuart M. Shieber. 1985. Using Restriction to Extend Parsing Algorithms for Complex-feature-based Formalisms. in *Proceedings of the 22nd Annual Meeting of the Association for Computational Linguistics, Chicago, Illinois, 8-12 July 1985*. University of Chicago, 145-152.

Stuart M. Shieber. 1986. *An introduction to unification-based approaches to grammar*. CSLI, Stanford California.

Melanie Siegel and Emily Bender. 2002. Efficient Deep Processing of Japanese. in *Proceedings of the 3rd Workshop on Asian Language Resources and International Standardization, 19th International Conference on Computational Linguistics (COLING 2002), Taipei, Taiwan*. Association for Computational Linguistics, Stroudsburg, Pennsylvania.

Gert Smolka. 1989. *Logic Programming over Polymorphically Order-Sorted Types*. Ph.D. Thesis, DFKI Saarbrucken.

Val Tannen, Peter Buneman, and Limsoon Wong. 1992. Naturally embedded query languages. in *Proceedings of the 4th International Conference on Database Theory (ICDT) 1992*, SpringerVerlag, London.

Hideto Tomabechi. 1991. Quasi-destructive graph unification. in *Proceedings of the 29th Annual Meeting of the Association for Computational Linguistics, Berkeley, California*. Association for Computational Linguistics, Stroudsburg, Pennsylvania.

Hideto Tomabechi. 1992. Quasi-destructive graph unifications with structure-sharing. in *Proceedings of the 15th International Conference on Computational Linguistics (COLING 1992), Nantes, France*. Association for Computational Linguistics, Stroudsburg, Pennsylvania.

Noriko Tomuro and Steven Lytinen. 1997. *Efficient Lazy Unification*. DePaul CTI Tech Report 97-006.

Marcel P. van Lohuizen. 1999. Parallel processing of natural language parsers. in *Proceedings of the International Conference on Parallel Computing (PARCO 1999), Delft University of Technology, Delft, Netherlands*.

Marcel P. van Lohuizen. 2000. Exploiting parallelism in unification-based parsing. in *Proceedings of the Sixth International Workshop on Parsing Technologies (IWPT 2000), Trento, Italy*.

Marcel P. van Lohuizen. 2000. Memory-efficient and Thread-safe Quasi-Destructive Graph Unification. in *Proceedings of the 38th Meeting of the Association for Computational Linguistics, Hong Kong, China, 2000*. Association for Computational Linguistics, Stroudsburg, Pennsylvania.

Marcel P. van Lohuizen. 2001. A generic approach to parallel chart parsing with an application to LinGO. in *Proceedings of the 39th Meeting of the Association for Computational Linguis-*

tics, Toulouse, France. Association for Computational Linguistics, Stroudsburg, Pennsylvania.

Marcel P. van Lohuizen. 2001. *Parallel Natural Language Parsing: From Analysis to Speedup*. Ph.D. Thesis. Technische Universiteit Delft.

David A. Wroblewski. 1987. Nondestructive graph unification. in *Proceedings of the 6th National Conference on Artificial Intelligence (AAAI-87)*. Morgan Kaufmann, 582-589.

Xianchao Wu, Takuya Matsuzaki, and Jun'ichi Tsujii. 2010. Fine-grained tree-to-string translation rule extraction. in *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics (ACL 2010)*. Association for Computational Linguistics, Stroudsburg, Pennsylvania, 325-334.

Yi Zhang, Rui Wang, and Yu Chen. 2011. Engineering a Deep HPSG for Mandarin Chinese. in *Proceedings of the 9th Workshop On Asian Language Resources*, Chiang Mai, Thailand, Association for Computational Linguistics, Stroudsburg, Pennsylvania.