Fachrichtung Informatik
Universität des Saarlandes

Efficient Parsing with Large-Scale Unification Grammars

Diplomarbeit

Ulrich Callmeier

Angefertigt unter der Leitung von
Prof. Dr. Hans Uszkoreit

Zweitkorrektur durch
Prof. Dr. Gert Smolka

Betreut von
Stephan Oepen, M.A.

März 2001

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

Saarbrücken, 26.3.2001

Ulrich Callmeier

Abstract

The efficiency problem in parsing with large-scale unification grammars, including implementations in the Head-driven Phrase Structure grammar (HPSG) framework, used to be a serious obstacle to their application in research and commercial settings. Over the past few years, however, significant progress in efficient processing has been achieved. Still, many of the proposed techniques were developed in isolation only, making comparison and the assessment of their combined potential difficult. Also, a number of techniques were never evaluated on large-scale grammars. This thesis sets out to improve this situation by reviewing, integrating, and evaluating a number of techniques for efficient unification-based parsing. A strong focus is set on efficient graph unification.

I provide an overview of previous work in this area of research, including the foundational algorithm in the work of Wroblewski (1987), for which I identify a previously unnoticed flaw, and provide a solution. I introduce the PET platform, which has been developed with two goals: (i) to serve as a flexible basis for research in efficient processing techniques, allowing precise empirical study and comparison of different approaches, and (ii) to provide an efficient run-time processor that supports fruitful scientific and practical utilization of HPSG grammars. The design and implementation of PET is presented in detail, including a closer look at efficient semi-lattice computation in the preprocessor.

A number of experiments with PET are discussed, using three existing large-scale HPSG grammars of English, Japanese, and German. I give precise empirical answers to some open research questions, most importantly the question of feature structure encoding (lists of feature-value pairs versus representations based on fixed arity), and show that this is a much less important factor than often assumed. I also address the question of predicting practical performance across grammars and processing platforms. Finally, I take a wider perspective and report on the overall improvement of processing performance for HPSG grammars (as exemplified by the LinGO grammar) that has been achieved over a period of four years by an international consortium of research groups.

Zusammenfassung

Das Effizienzproblem beim Parsen großer Unifikationsgrammatiken, z. B. Implementationen im Rahmen der *Head-driven Phrase Structure Grammar* (HPSG), war lange Zeit ein ernsthaftes Hindernis für deren Benutzung in der Forschung und in praktischen Anwendungen. In den letzten Jahren wurde jedoch ein signifikanter Fortschritt in der effizienten Verarbeitung erzielt. Allerdings existierten viele der vorgeschlagenen Verfahren nur isoliert, was ihren Vergleich und die Einschätzung ihres kombinierten Potentials nur schwer möglich machte. Zudem ist eine Reihe von Verfahren nie auf großen Grammatiken evaluiert worden. Diese Arbeit setzt sich zum Ziel, diese Situation zu verbessern, indem Techniken zum effizienten unifikationsbasierten Parsen begutachtet, integriert und evaluiert werden. Der Schwerpunkt liegt dabei auf effizienter Graphunifikation.

Ich gebe einen Überblick über vorhergehende Arbeiten auf diesem Gebiet, einschließlich des grundlegenden Algorithmus aus der Arbeit von Wroblewski (1987), für den ich einen Fehler aufdecke und eine Lösung präsentiere. Ich stelle die PET-Plattform vor, die mit zwei Zielen entwickelt wurde, zum einen, um als eine flexible Basis für die Untersuchung effizienter Verarbeitungsstrategien zu dienen, die präzise empirische Untersuchungen und den Vergleich verschiedener Ansätze ermöglicht, zum anderen, um ein effizientes Laufzeitsystem zur Verfügung zu stellen, das die fruchtbare wissenschaftliche und praktische Nutzung von HPSG-Grammatiken ermöglicht. Ich stelle das Design und die Implementation von PET vor, wobei ich genauer auf die effiziente Implementierung der *semi-lattice* Berechnung im Präprozessor eingehe.

Ich diskutiere eine Reihe von Experimenten mit PET, die mit drei existierenden großen HPSG-Grammatiken für Englisch, Japanisch und Deutsch durchgeführt werden. Dabei gebe ich präzise empirische Antworten auf einige offene Fragen des Gebiets, insbesondere auf die Frage der Kodierung von Merkmalsstrukturen (Listen von Attribut-Wert-Paaren gegenüber Repräsentationen, die auf fester Arität beruhen), und zeige, dass dies viel weniger Bedeutung hat als oft angenommen. Ich gehe auch der Frage nach, inwiefern Performanz in der Praxis über die Grenzen von Grammatiken und Verarbeitungssystemen hinweg vorhergesagt werden kann. Schließlich betrachte ich aus einer umfassenderen Perspektive den Gesamtfortschritt in der Verarbeitung von HPSG-Grammatiken über einen Zeitraum von vier Jahren anhand der LinGO-Grammatik; dieser Fortschritt wurde in einem internationalen Konsortium von Forschungsgruppen erzielt.

Acknowledgments

I am indebted to Stephan Oepen for advice, fruitful discussions, help with the [incr tsdb()] profiling environment, and his significant contribution to the conception of the Tomabechi variant from Section 4.6.2, the partitioning algorithm in Section 4.6.3, and the discussion in Section 4.8. I am grateful to Dan Flickinger, Ann Copestake, and John Carroll for a number of inspiring discussions and answers to numerous questions about the LinGO grammar and the LKB system. Melanie Siegel and Stefan Müller were most helpful in all questions regarding the Japanese and German grammars, respectively. I would like to thank Liviu Ciortuz for the experience that I gained while working with him on the CHIC system at DFKI Saarbrücken. Marcel van Lohuizen was a source of a number of improvement ideas for the preprocessor output, and other aspects of PET. I thank Anne Wagner for contributing Figure 3.1, and Flix (www.der-flix.de) for designing and realizing the cover, and for the drawing on the back cover. Special thanks go to Stephan Oepen, Dan Flickinger, Tobias Callmeier, Peter Wagner and Axel Culmsee for reading and commenting on parts or all of this thesis. Finally, I would like to thank my professor, Hans Uszkoreit, for providing the stimulating environment where this research could be carried out.

Contents

1. Introduction **1**
1.1. Outline . . . 3

2. Previous and Related Work **5**
2.1. Kartunnen and Kay's Structure Sharing with Binary Trees . . . 5
2.2. Pereira's Structure Sharing Representation . . . 6
2.3. Karttunen's Reversible Unification . . . 7
2.4. Wroblewski's Nondestructive Graph Unification . . . 8
2.5. Godden's Lazy Unification . . . 14
2.6. Kogure's Strategic Lazy Incremental Copy Graph Unification . . . 15
2.7. Emele's Unification with Lazy Non-Redundant Copying . . . 17
2.8. Tomabechi's Quasi-Destructive Graph Unification . . . 18
2.9. Extensions to Tomabechi's Algorithm . . . 19
2.10. Van Lohuizen's Variant of Quasi-Destructive Graph Unification . . . 20
2.11. Summary . . . 21
2.12. Alternative Approaches . . . 22

3. The PET System **25**
3.1. Overview . . . 25
3.2. The Preprocessor . . . 27
3.3. The Parser . . . 30

4. Empirical Results **35**
4.1. Experimental Setup . . . 36
4.2. Setting Out: A Closer Look at Copying . . . 39
4.3. Zooming In: How the Parser Spends its Time . . . 42
4.4. Fine Tuning the Quick Check Filtering Method . . . 45
4.5. Partial Expansion and Unfilling . . . 47
4.6. Feature Structure Encoding Techniques . . . 49
4.7. Copying, Recomputation and Trailing in Active Chart Parsing . . . 54

4.8. Cross-comparison: Comparing Across Grammars and Platforms 57

5. Conclusions and Outlook **61**
5.1. Summary 61
5.2. Quantifying Progress 62
5.3. Open Questions and Future Work 63

A. The Input Language **65**

B. Binary Representation of Grammars **69**

C. PET Usage **75**

D. Virtual Appendix **83**

Bibliography **85**

Index **95**

1. Introduction

Most non-transformational linguistic frameworks such as LFG (Bresnan and Kaplan, 1982; Dalrymple et al., 1995), GPSG (Gazdar et al., 1985), and HPSG (Pollard and Sag, 1987, 1994) have a common property: They use a data structure called a *feature structure* to represent linguistic knowledge, and all these frameworks are based on the central operation of *unification* of feature structures[1]. The efficiency of any processing system based on such formalisms crucially depends on the efficiency of the unification operation. This becomes even more significant for strongly lexicalized frameworks such as HPSG, where feature structures tend to carry a relatively large amount of information[2]. Efficiency here does not only refer to speed, but also, maybe even more importantly, to memory consumption. Feature structure representation and the choice of unification algorithm play a central role in the memory consumption characteristics of any unification-based processing system. While there are other important factors for the overall performance of a unification-based processing system besides the efficiency of feature structure unification (like the choice of parsing strategy, or the amount of money you can spend on a fast machine with lots of memory), without an efficient implementation of this basic operation, satisfactory overall performance cannot be expected.

Until only a few years ago, time and memory requirements of unification-based processing systems used to be prohibitive for many applications. Several hundred megabytes of main memory were considered the absolute minimum for parsing medium-complexity input with a large-scale grammar, and parsing times of one or two minutes per sentence were not considered unusual. Over the past few years, however, significant progress in efficient processing has been achieved. In a collaborative research effort, a consortium of groups at Saarbrücken[3] (Uszkoreit et al., 1994; Krieger and Schäfer, 1994a), CSLI

[1] For an introduction to unification-based grammars, see Shieber (1986). For a formal discussion of typed feature structures, see Carpenter (1992b); an excellent updated discussion can be found in Penn (2000). For the details of the formalism underlying this work, see Copestake (2000).

[2] For instance, in LinGO, an implementation of a HPSG grammar of English, the average size of feature structures when parsing sentences of medium complexity (i.e. sentences of length 10 to 20 words) is around 300 nodes, see Table 4.6 on page 49.

[3] See `http://www.dfki.de/lt/` and `http://www.coli.uni-sb.de/` for information on the DFKI Language Technology Laboratory and the Computational Linguistics Department at Saarland University, respectively.

Stanford[4] (Copestake, 1992; Flickinger and Sag, 1998; Copestake and Flickinger, 2000), the University of Tokyo[5] (Torisawa and Tsujii, 1996; Makino et al., 1998; Tateisi et al., 1998), and other groups, have worked on the development of large-scale HPSG grammars, grammar engineering platforms, and efficient processors. For more information on the collaborative effort and its results, see Flickinger et al. (2000b); Oepen and Callmeier (2000); Oepen et al. (2001). This thesis was written in the context of this collaborative effort. Some of the results presented here have been published or submitted for publication before (Callmeier, 2000; Oepen and Callmeier, 2000; Callmeier, 2001); the work on hand contains extended and updated presentations of these results.

When starting the preparations for this thesis in 1999, many of the proposed techniques for efficient unification-based processing existed only in isolation, making comparison and an assessment of their combined potential difficult at best. A number of algorithms and techniques had never been evaluated on large-scale grammars. Even if an evaluation on a large-scale grammar had taken place, it was often restricted to one particular grammar, which made it difficult to estimate the applicability for different grammars. Also, two of the existing processing systems[6] in the consortium, PAGE (DFKI Saarbrücken) and the LKB (CSLI Stanford), primarily designed as grammar development platforms, were still not efficient enough to support empirical evaluation of different processing strategies on multiple grammars and large sets of data in reasonable time, nor of the use of large-scale grammars in practical application contexts. One of the reasons for this was the underlying implementation language of both platforms, namely Lisp, which incurs a considerable overhead in memory management cost for this kind of application, since in unification-based processing huge amounts of data are allocated and released in short time. In addition, while many of the techniques developed for efficient processing were implemented in both platforms, each platform implemented some techniques that were missing in the other, and some techniques were implemented in neither of the two.

This thesis sets out to improve this situation by reviewing, integrating and evaluating a number of techniques for efficient unification-based parsing, with a strong focus on efficient graph unification. Since no analytical tools exist that could predict how the properties of individual unification-based grammars will interact with particular processing techniques (see Carroll (1994) for discussion), empirical study is indispensible for the evaluation and optimization of such techniques. To obtain meaningful results, empirical study requires controlled experiments on large sets of data. I implemented the PET platform (presented in Chapter 3) to serve as a flexible basis for research in processing techniques, allowing precise empirical study and comparison of different approaches. A

[4] The `http://lingo.stanford.edu/` web pages list HPSG-related projects and people involved at CSLI, and also provide an online demonstration of the LKB system and LinGO grammar.

[5] Information on the Tokyo laboratory can be found at `http://www.is.s.u-tokyo.ac.uk/`

[6] A third platform in the consortium, the LiLFeS system (University of Tokyo), is based on compilation to an abstract machine. See Section 2.12 for more information.

second, orthogonal goal was pursued, namely to provide an efficient run-time processing system that supports fruitful scientific and practical utilization of HPSG grammars, complementing the existing development platforms PAGE and LKB.

The approach of systematic experimentation and precise empirical study of system properties is called *competence & performance profiling* (Oepen, 2001; Oepen and Carroll, 2000b; Oepen and Callmeier, 2000; Oepen and Flickinger, 1998). Suitable tools are a key requirement for fruitful systematic experimentation and comparison. I use the [incr tsdb()][7] environment for my experiments. [incr tsdb()] provides facilities to obtain, analyze, and compare rich, precise, and structured snapshots of system behavior (called *profiles*). It defines a set of descriptive metrics that aim both for in-depth precision and generality across different processing systems, enabling their comparison. Section 4.1 provides more details about the role of [incr tsdb()] in my work.

Situated in the context of the aforementioned collaborative research effort, I was able to base my experimental work on three different large-scale HPSG grammars, the LinGO grammar of English (Flickinger et al., 2000a; Copestake and Flickinger, 2000), the Japanese Verb*mobil* grammar (Siegel, 2000), and the Verb*mobil* grammar of German (Müller, 1999; Müller and Kasper, 2000). These three grammars obey a common descriptive formalism that the consortium has converged on (Oepen et al., 2000). It can be characterized as a conservative blend of Carpenter (1992b), Copestake (1992), and Krieger and Schäfer (1994a), providing a closed-world, conjunctive-only, multiple inheritance type system that enforces strong typing and strict appropriateness, and allows types to be associated with complex constraints that are inherited and applied at run time[8]. All processing systems within the consortium, including PET, implement this core formalism, sometimes enriched by platform-specific extensions (e.g. default unification in the LKB).

1.1. Outline

The organization of the thesis is as follows. In Chapter 2, I provide a review of previous work, discussing a problem in Wroblewski (1987) and identifying open questions. Chapter 3 moves on to a description of PET, my experimentation platform and run-time system. I discuss the architecture and implementation of the two main components of PET, viz. the preprocessor and the parser, and take a closer look at the implementation of semi-lattice construction in the preprocessor. Chapter 4 describes a series of experiments that I carried out using PET and answers a number of the open questions identified in Chapter 2. Highlights include a comparison of three alternative feature structure encodings (one based on lists of features and values, the other two exploiting fixed arity), and a comparison of two

[7] The (draft) [incr tsdb()] user manual, the software package itself, and an explanation of the strange-looking name can be obtained from `http://www.coli.uni-sb.de/itsdb`

[8] For details, see Copestake (2000).

processing systems (the LKB and PET) across the three grammars. I conclude with a summary of the results, a review of progress in processing the LinGO grammar over a period of four years, and a set of open questions for further work (Chapter 5).

2. Previous and Related Work

This chapter reviews previous related work on efficient graph unification in natural language processing. Graph unification experienced a wave of interest from the mid-eighties to the early nineties. I discuss the relevant literature in Sections 2.1 to 2.8. Recent contributions to this line of research are reviewed in Sections 2.9 and 2.10. I summarize the various approaches in Section 2.11, and identify a number of open questions, that are answered in Chapter 4. Finally, I provide a brief survey of alternative approaches.

The papers that I discuss in the first two sections both date back to 1985, and are the earliest published works on efficiency issues in graph unification in the computational linguistics literature. Both Karttunen and Kay (1985), which is the subject of the next section, and Pereira (1985) identify copying as an expensive operation that can be partially avoided. The common solution proposed is structure sharing[1], albeit by two different means.

2.1. Kartunnen and Kay's Structure Sharing with Binary Trees

Karttunen and Kay (1985) propose representing feature structures as binary trees. Using binary trees to represent feature structures allows a simple addressing scheme that can be exploited for a structure sharing representation. The addressing scheme assigns an index to nodes that corresponds to the concatenation of a binary representation of the turns one takes to reach this node from the root of the tree. A left turn is represented by 0 and a right turn by 1. Minimizing copying is achieved by duplicating only changed parts of a graph, i.e. only the topmost node of a tree is copied as well as nodes along any paths leading to changed nodes. This is the very same idea used in the structure sharing schemes proposed later in Tomabechi (1992) and Malouf et al. (2000). The idea of lazy copying

[1]When talking about structure sharing here and in the following, I do not refer to the structure sharing that represents reentrancies in one graph, but rather to the sharing of subgraphs between two distinct graphs. Tomabechi (1992) calls the first kind *feature-structure sharing* and the second kind *data-structure sharing*. Malouf et al. (2000) call the second kind of structure sharing *subgraph sharing*.

is also found in a number of later papers (Godden, 1990; Kogure, 1990; Emele, 1991; Tomabechi, 1991).

One difficulty with this scheme is to maintain the trees balanced. Without further provisions, the trees could degenerate, resulting in trees with access times much worse than the logarithmic one in the balanced case. Karttunen and Kay (1985) propose a solution to this problem; during unification, whenever we have to choose which tree should embed another tree, we choose the combination that results in the combined tree of minimal height. To implement this efficiently, we have to maintain information about the shortest path from each node down to a node with a free left or right pointer. This allows to efficiently find the shallowest place in a tree to embed another one. To easily determine the shallowest combined tree, we also have to store the length of the longest path originating from each given node.

No evaluation of the method is given, nor details about an implementation of this scheme. It remains questionable if the inevitable overhead of encoding a feature structure in a binary tree can be compensated by the gain from structure sharing. Also, it is not clear if the proposed solution to the problem of the binary trees becoming imbalanced is sufficient, because for large structures considerable imbalances can still occur.

Karttunen and Kay make an interesting remark:

> *In a sense, most of the copying effort is wasted. Unifications that fail typically fail for a simple reason. If it were known in advance what aspects of structures are relevant in a particular case, some effort could be saved by first considering only the crucial features of the input. (Karttunen and Kay, 1985, p. 133)*

This is exactly what is later successfully realized in the quick check filtering technique (Malouf et al., 2000; Kiefer et al., 1999), discussed in Section 4.4.

2.2. Pereira's Structure Sharing Representation

Another important early research effort into efficient graph unification is documented in Pereira (1985). Pereira identifies copying as a source of inefficency, and proposes to eliminate most of this copying by representing updates to objects separate from the objects themselves. This is inspired by structure sharing methods from theorem proving (Boyer and Moore, 1972) and Prolog implementation (Warren, 1977).

The basic idea of Pereira's approach is that an initial object, together with a list of update records, contains the same information as the object that results from applying the updates to the initial object. In this way, the cost of applying the updates (with possible copying to avoid destructively modifying the original object) can be traded for the cost of having to compute the effects of updates when accessing the derived object. In Pereira's approach, dag instances are represented by a skeleton (the inital dag) and an environment

(a table of updates). Pereira suggests to use virtual-copy arrays known from Prolog implementation (Warren, 1983b) to represent dags. This gives the benefit of an $O(\log n)$ access and update time for an array with highest used index n, while old contents are preserved when updating.

This allows two levels of sharing: The Boyer-Moore style dag representation allows derived dag instances to share input data structures (skeletons), and the virtual-copy array environment representation allows different branches of the search space to share update records. The price to pay is an $O(\log d)$ overhead (where d is the number of nodes) in any graph access.

Pereira reports about preliminary tests with a Prolog implementation of his scheme on various grammars, where the structure sharing scheme is up to 60% (and never less than 40%) faster than the same parsing algorithm with structure copying. No details about the grammar or the input used is given. Without information on the size of the feature structures and the size of the parse chart it is hard to estimate how much the node access overhead and the cost of applying environments affects parsing time. It could be expected that this cost becomes more significant for large feature structures, and deeply embedded skeleton/environment structures necessitating frequent reapplication of environments. I describe a related experiment investigating the efficiency of a skeleton/environment scheme in Section 4.7.

2.3. Karttunen's Reversible Unification

Karttunen (1986) identifies destructive unification and the necessitated copying as a source of computational inefficiency. Karttunen proposes a novel solution to this problem, by implementing unification in a way so that the original state of the input structures can be restored after unification has been completed. This is achieved by saving original attribute values together with the attribute-value pair, whenever a destructive change is about to be made. In this way, the effects of unification can be undone by restoring the original values of all saved attribute-value pairs.

Karttunen gives no details of how his scheme is implemented. No evaluation is presented either, although Karttunen remarks:

> *It appears that this simple technique is more effective in reducing parsing time than the methods discussed in the literature [...] that improve the efficiency of copying by using structure sharing.* (Karttunen, 1986, p. 22)

What Karttunen overlooks here, is that structure sharing is also possible with a reversible unification scheme.

Karttunen's basic idea is the very same that is later suggested in Tomabechi (1991), with the important improvement of integrating a mechanism from Wroblewski (1987) that

allows undoing the modifications in constant time. Tomabechi (1992) and Malouf et al. (2000) integrate structure sharing into this approach.

2.4. Wroblewski's Nondestructive Graph Unification

Wroblewski (1987) starts out from the assumption that the fundamental problem with previous approaches to graph unification is implementing it as a destructive operation, because this necessitates copying, which has been identified as a computational sink. To remedy this, Wroblewski proposes to use a nondestructive algorithm. While this has been proposed before (Karttunen, 1986), Wroblewski's important contribution is the definition of *over copying* and *early copying*. Although the latter has been redefined later by Tomabechi (1991) (see Section 2.8), both definitions have been used troughout the work in this area since then (Godden, 1990; Kogure, 1990; Tomabechi, 1991; Emele, 1991; Tomabechi, 1992, 1995).

Wroblewski classifies the fraction of copying that is done unnecessarily in two parts. There can be too much copying when copies are made of both input dags, and then both are destructively modified by the unification operation. In this way, the raw material for two input dags is used to create just one new dag, whose size almost never equals the sum of the two input dag sizes. Wroblewski calls this type of unnecessary copying *over copying*. A better algorithm would only allocate enough memory for the resulting dag. Another way of unnecessary copying is when the input dags are copied before unification starts, but the input dags later turn out not to be unifiable. This kind of unnecessary copying is called *early copying* by Wroblewski. A better algorithm would check for unifiability first, and only copy for compatible graphs. Wroblewski observes that over copying and early copying are independent properties of a unification algorithm, and thus can be treated independently.

The solution proposed by Wroblewski is to implement unification nondestructively, by incrementally copying the argument graphs. This requires additional slots in the graphs to keep track of the copies associated with each node. A problem with his algorithm, identified by Wroblewski, is that it still over copies in certain cases, namely when copies of nodes that are found to be reentrant later, were previously made independently.

Wroblewski compares his scheme with Pereira (1985) and Karttunen (1986). He acknowledges that Pereira's scheme would not over copy in cases where his own scheme would. He adds that he found the skeleton/environment solution hard to implement, and most of its speed advantage was cancelled by the $\log(d)$ node access overhead. Compared to Karttunen's scheme, Wroblewski sees an important advantage in the fact that his algorithm can undo changes (in the *copy* slots) in constant time by incrementing a global

counter and protecting *copy* slot access through a validity marker. This idea (attributed to Mark Tariton) later plays a central role in Tomabechi's quasi-destructive unification (Tomabechi, 1991).

Wroblewski identifies four tradeoffs in implementing graph unification, namely over copying, early copying, dag access overhead and restrictiveness to certain contexts. He also suggests future research into techniques to intelligently eliminate rule application without invocing unification; such techniques have later been presented in Kiefer et al. (1999) and Malouf et al. (2000).

Wroblewski does not present an evaluation of his algorithm. Also, no information is given on how much over copying and early copying occurs in practice. I present an evaluation of the algorithm in Section 4.2, and also present detailed numbers on over copying and early copying with a large scale grammar.

2.4.1. Problems with the Algorithm

In spite of the long time Wroblewski's algorithm is known, and although it has served as a starting point for improved algorithms in a number of later papers (Godden, 1990; Kogure, 1990; Emele, 1991; Tomabechi, 1991), none of these papers mention the problem that I discuss in this section.

The procedure `unify2` (see Figure 2.1) as described in Wroblewski (1987) is not correct. The problem arises from incorrectly handling the case where a copy exists for one of the nodes, but not for the other (line 23 in Figure 2.1). In this case, a special version of `unify1` is invoced on the copy and on the node that does not have a copy. This special version of `unify1` preserves its second argument and records all changes in the first argument.

There are, however, cases where this is not possible. One such case is illustrated in Figure 2.2. Here, `unify2` has been recursively invoced on the nodes ④ and ⑦, following the arcs labelled `C`. There is a copy already associated with ⑦ in the right graph, since it was previously visited when unifying ② and ⑦ following the arcs labelled `A`. There is no copy associated with ④. Thus, the condition on line 21 in Figure 2.1 is true, and `unify1` is called on ④ and ⑨, the copy of ⑦.

There is no possible way for `unify1` to produce the correct result node (a node with two arcs labelled `B` and `D`) without destructively modifying ④, which is part of the original input graph and must not be modified. Since `unify1` does not make copies, one of the input nodes will be the representative of the result. If ④ is chosen as the representative, an arc labelled `B` has to be added, thus modifying the node; if ⑨ is chosen as the result representative, ④ has to be forwarded to ⑨, thus modifying ④. One might argue that a solution to this problem is representing the result in ⑨, which can be modified, and not forwarding ④. In fact, Wroblewski (1987) gives this explanation for line 23 in Figure 2.1:

```
1   PROCEDURE Unify2 (d1 d2)
2     Dereference d1, d2.
3     IF neither d1 nor d2 have copies THEN
4       copy = a new node. copy.status = "copy".
5       d1.copy, d2.copy = copy.
6       newd1 = complementarcs(d1, d2).
7       newd2 = complementarcs(d2, d1).
8       shared = intersectarcs(d1, d2).
9       FOR all arcs in shared DO
10        Find the corresponding arc in d2.
11        Recursively unify2 the arc values.
12        IF unify2 failed THEN
13          Return failure.
14        ELSE
15          Add a new arc in copy.
16        ENDIF
17      FOR arc in union(newd1, newd2) DO
18        Copy the arc-value of each arc, honoring existing
19        copies within, and place the value in copy.
20      Return Copy.
21    ELSE if d1 xor d2 has a copy THEN
22      Without loss of generality, assume d1 has the copy.
23  *   unify1(d1.copy, d2) preserving d2.
24      Return d1.copy.
25    ELSE if both d1 and d2 have copies THEN
26      Unify1(d1.copy, d2.copy).
27    ENDIF
28  ENDPROCEDURE
```

Figure 2.1.: The procedure `unify2` from Wroblewski (1987)

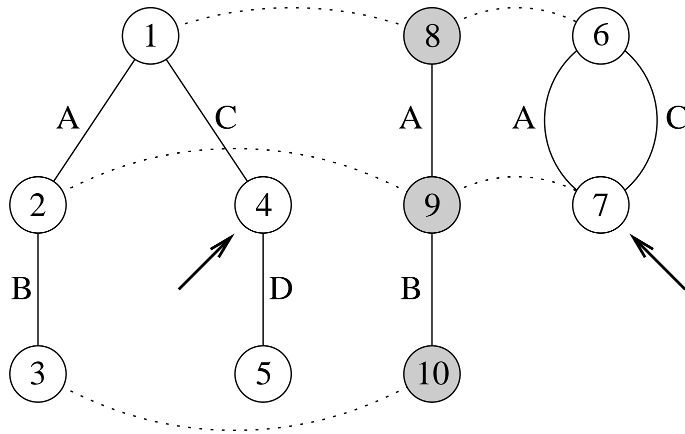

Figure 2.2.: Problematic situation for Wroblewski's algorithm. `unify2` is currently working on the two nodes pointed two by the arrows. Grey nodes represent copies. Dotted lines represent pointers in the *copy* slots.

```
21      ELSE if d1 xor d2 has a copy THEN
22        Without loss of generality, assume d1 has the copy.
23   +    d2.copy = d1.copy.
24        unify1(d1.copy, d2) preserving d2.
25        Return d1.copy.
```

Figure 2.3.: Fix for procedure `unify2` from Wroblewski (1987). Line 23 has been added.

> *When a copy already exists for one graph or the other, but not both, this algorithm will perform an operation very much like* `unify1`*, but no forwarding will be done since the changes can all be safely recorded in the copy. This is what is meant by the line marked with an asterisk. (Wroblewski, 1987, p. 584)*

This might seem to work at a first glance, but it is not a correct solution either, as potential reentrancies with ④ will be lost: if we encounter ④ again during this unification, we do not know that it has been unified with ⑦ already and the result has been represented in ⑨. A better solution becomes obvious (and was probably intended by Wroblewski): we forward ④ to ⑨ using the *copy* slot. The fix is shown in Figure 2.3.

Even with this fix in place, `unify2` remains problematic. Figure 2.4 shows the situation from above after `unify2` has finished its work on ④ and ⑦, and thus finished processing the input graphs. The problem here is that the result graph returned by `unify2` shares ⑤ with the left input graph, because `unify1` is called to unify ④ and ⑨, but `unify1` by design never copies. Thus, the result is not independent of the input graphs. This can cause problems when graphs are modified by operations other than unification,

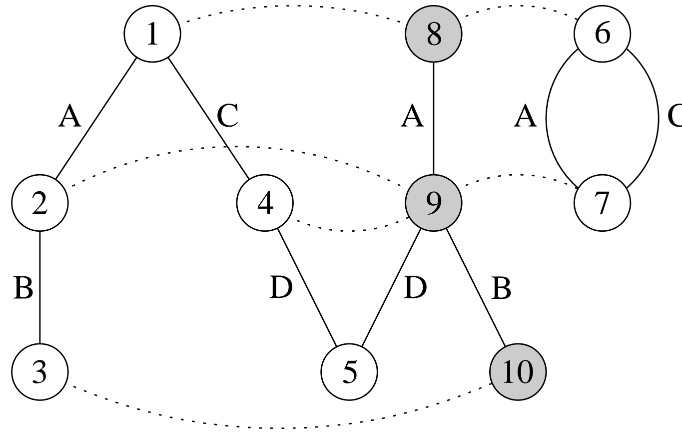

Figure 2.4.: Situation from above after the fixed `unify2` has finished processing the input graphs. Node ⑤ has not been copied.

e.g. when removing parts of a graph using a restrictor. Also, since coindexation is represented via graph reentrancies, it can result in spurious coindexation (e.g. when using one lexical entry multiple times in parsing one sentence, Malouf et al. (2000) discuss the problem in more detail). Note that independent of my fix, this problem also arises in simpler situations. If in the example above ② did not have any outgoing arcs, we could have just forwarded ⑨ to ④ — still, ⑤ would end up uncopied in the result.

A related problem presents itself when trying to adapt `unify2` for typed unification. Here, situations as described above, where `unify1` is not able to represent the result without modifying one of the input graphs occur more frequently, namely when the greatest lower bound type of the two unified nodes is different from each of the two input types. Again, `unify1` is called when one of the nodes already has a copy, but it cannot represent the result without creating a new copy.

2.4.2. An Improved Version of `unify2`

There is no trivial fix to solve the problems described in the last section. It seems for cases like the ones presented, we need a cross between `unify1` and `unify2`, i.e. a unification procedure that is allowed to destructively modify one of its arguments, but not the other. If such a function is used instead of `unify1` in places where one node is a copy, but not the other, the problems detailed in the last section are solved. I call this function `unify3`.

The change to `unify2` itself is given in Figure 2.5, the new function `unify3` is shown in Figure 2.6. `unify3` is relatively straightforward. First, the arguments need to be dereferenced (line 2), if they are identical (a reentrancy), the first node (d_1) is returned as the result representative (line 3). Then two cases need to be considered: d_1 can have

```
21     ELSE if d1 xor d2 has a copy THEN
22       Without loss of generality, assume d1 has the copy.
23       Return unify3(d2, d1.copy).
```

Figure 2.5.: Final fix for procedure `unify2` from Wroblewski (1987).

```
1   PROCEDURE unify3 (d1, d2) (* Unify d1 into d2 *)
2     Dereference d1, d2.
3     IF d1 and d2 are identical THEN Return d1.
4     IF d1 has no copy THEN
5       IF d1 is a copy THEN Return unify1(d1, d2).
6       d1.copy = d2.
7       FOR each arc a1 in d1
8         IF d2 has a corresponding arc a2
9           unify3(a1.value, a2.value)
10          IF unify3 failed THEN Return failure.
11        ELSE
12          Copy a1.value honoring existing copies within.
13          Add a new arc with the copied value to d2.
14        ENDIF
15      Return d2.
16    ELSE
17      Return unify1(d1.copy, d2).
18    ENDIF
19  ENDPROCEDURE
```

Figure 2.6.: New procedure `unify3`. The first argument graph cannot be modified, the second one may be changed.

a copy already, then `unify1` can be used to destructively unify the copy of d_1 and the second argument d_2 (line 17). Otherwise, we need to check if d_1 is a copy (using the *status* field), in this case `unify1` can be used (line 5). Otherwise, d_2 is registered as the copy of d_1 (line 6), and we go through all arcs of d_1. Whenever a corresponding arc exists in d_2, we call `unify3` on the arc values (line 9). When a corresponding arc does not exist, a copy is inserted into d_2 (line 12–13).

I do not give a formal correctness proof, but I will briefly discuss why `unify3` works as intended. It is important to keep three invariants in mind that are assumed throughout:

1. `unify3` never destructively modifies its first argument node, nor the nodes below[2].

2. The second argument to `unify3` is a copy.

3. When a node is a copy, all nodes below it are copies.

We can see by inspection that `unify3` maintains these invariants, the second and third invariant are also maintained by `unify2`. This is enough to show that `unify3` behaves as expected with respect to not destructively modifying its first argument. I do not show here that it actually performs unification of the argument dags.

2.5. Godden's Lazy Unification

Godden (1990) presents a new solution to the efficiency problem with copying argument graphs before destructive unification. It requires no new slots in the structure of nodes and only minor revisions to the unification algorithm. The basic idea is using lazy evaluation techniques to delay copying — Godden argues that lazy evaluation is a good technique for the copying problem in graph unification because the overwhelming majority of copying is unnecessary. Dags are turned into an active data structure, using two basic procedures: *delay* and *force*. In this way, copying can be *delayed* until it is *forced*. When copying is unnecessary, it is never *forced*, and thus *delayed* infinitely. Godden implements *delay* by using *closures* provided by the programming language.

Adapting an existing unification algorithm for lazy evaluation requires only minimal modifications. Whenever a destructive operation is about to be made on a node, a procedure called `force-delayed-copy` is called on this node first. This procedure forces the next level of any delayed copying of this node, subsequently delaying the next level of copying. This scheme reduces copying both in the case of unification failure, and in the case of a successful unification, for two reasons. On the one hand, it uses only the minimal required number of nodes (not the sum of the nodes of the two input graphs), and on the

[2] I say a given node is *below* another node in a feature structure, if there is a sequence of attribute-value pairs through which the latter can be reached from the first.

other hand it often needs even fewer nodes, when nodes do not need to be copied at all. The second kind of unnecessary copying is called *redundant* copying in Kogure (1990), and avoided through a lazy copying scheme. Godden's algorithm needs two more refinements: it needs to remember if a given node has already been copied. This is done using *copy environments*. These must be associated to individual dags, and cannot be global, because several dags may need to be preserved during parsing. The second refinement is for nodes that remain delayed after successful unification. These nodes can acquire a new root by subsequent unification, and then attempted to be copied from inside this new root. The solution is to give the node access to the new copy environment, this can be done by destructively merging copy environments.

An empirical evaluation of lazy unification is provided for ten sentences parsed by the TASLINK natural language system (Godden, 1990). Lazy unification copies only 7% of the nodes copied by eager unification — however, this does not take into account the cost of creating closures to delay copying. The actual speedup is about 50%. Godden identifies several sources of inefficiency in his implementation of lazy unification that could be fixed easily, thus further improving performance of his method. Godden concludes with an interesting remark: while he considers the end result as elegant, development was slow due to the difficulties lazy evaluation presents for debugging.

While Godden's solution for the copying problem is conceptually elegant, part of the cost of copying is merely hidden in programming language specific constructs for delaying operations. The potentially costly creation of closures is unnecessary in cases where those closures remain unchanged in the final result. Also, the search in the copy environments and merging of those environments causes additional overhead. The limited evaluation provided by Godden makes it hard to estimate the benefits of lazy unification on input data with different characteristics than Godden's data. Also, a comparison to Wroblewski's method is missing, which already creates significantly fewer nodes than an eager unification method, so it remains an open question if the overhead of delaying copying pays off compared to Wroblewski's approach.

2.6. Kogure's Strategic Lazy Incremental Copy Graph Unification

Two methods for improving the efficiency of feature structure unification are presented in Kogure (1990). The first is lazy incremental copy graph unification (*LING*), a novel method to achieve structure sharing while maintaining constant time node access. The second method is strategic incremental copy graph unification (*SING*), which implements an early failure finding strategy which tries to unify substructures that tend to fail in unification first. The two methods can be combined (*SLING*).

Kogure introduces the term *redundant copying* for the unnecessary copying that takes place when, instead of sharing structure, unchanged parts of the input graphs are copied. Kogure identifies redundant copying as the main deficiency of the algorithm suggested in Wroblewski (1987), because in Wroblewski's method the result graph consists of newly created nodes only.

The lazy incremental copy graph unification method presented by Kogure is an extension of `unify2` from Wroblewski (1987). A lazy evaluation technique is used instead of completely copying arcs without a counterpart in the corresponding structure. In this way, copying a node is delayed until either its own contents need to change, or until a node below it needs to be copied. To implement this, the procedure to copy a node is revised to maintain copy dependency information. The node itself is not copied immediately, through the copy dependency information we know what other nodes need to be copied if this node is modified later.

The strategic incremental copy graph unification method is based on learning information about which parts of graphs are more likely to fail than others. It takes advantage of the fact that unification tends to fail often under some features, and only rarely under others (e.g. semantics construction in a parser). The frequencies need to be learned since they are application specific, e.g. in a generator unification of semantics will fail often. The learning process uses randomized feature treatment orders. Failure tendency is recorded for type-feature pairs. This learned failure tendency information is then used to apply unification in an order that first treats features with the highest tendency to fail, by ordering the arcs that are recursively unified (shared arcs) in each unification step.

Unfortunately, Kogure presents no evaluation of his techniques. Thus, it is hard to estimate their effectiveness. Both techniques incur an overhead that can potentially be significant. The *LING* method requires maintaining the copy dependency information, and actual copying requires a second traversal of the affected nodes; the *SING* method requires to sort shared arcs according to failure frequency in each step (or maintain this order at all times), this can be costly, as discussed in Tomabechi (1995), and verified by my own experiments. Like any incremental copying technique, *LING* suffers from early copying, see Section 2.8.

Kogure's idea of using failure frequency information was later taken up and generalized by Uszkoreit (1991). Failure frequency information is also crucial in the quick check filtering technique (Malouf et al., 2000; Kiefer et al., 1999). I take a closer look at the quick check in Section 4.4.

2.7. Emele's Unification with Lazy Non-Redundant Copying

A new attempt at solving the copying problem is presented in Emele (1991). As in Kogure (1990), the problem of redundant copying (see Section 2.6) is identified as the main weakness of previous approaches. The solution synthesizes ideas of lazy copying with the notion of chronological dereferencing to achieve a high amount of structure sharing. Emele criticizes Godden (1990) and Kogure (1990) for having to copy all the nodes on a path leading to a node that needs to be copied, even if these nodes are not changed themselves. This can incur a considerable cost, since such cases are common in unification-based parsing. Also, both methods require the copying of arcs to a certain extent. Curiously, in previous discussions of the copying problem, the copying of arcs was simply ignored.

Emele's Lazy Incremental Copying (LIC) method is based on Wroblewski's idea of incrementally producing the result dag during unification, leaving the argument dags untouched. Copied nodes are associated with the input structure's nodes by means of a *copy* slot in the node representation. Unlike Wroblewski, however, copies are created lazily. Only in cases where an update to a node of one of the input structures leads to a destructive change copying is required. In this way, Emele claims to combine the advantages of Wroblewski's incremental copying and structure sharing from Pereira (1985), avoiding the disadvantages of both methods. Instead of using global environments as Pereira, in the LIC approach each node records its own updates in the copy field and a generation counter. This makes complex merging of environments unnecessary, but requires a new dereferencing operation. Usually, dereferencing follows a pointer chain all the way to the end to find the destination node. In LIC, however, dereferencing is performed according to an environment. A generation counter is associated to each "copynode", indicating the generation to which it belongs. An environment is just a sequence of generation counters. Dereferencing follows pointers only as long as the destination node's generation is older than the youngest generation in the environment. Environments are extended with a new generation whenever a choice-point is encountered, the length of the environment corresponds to the number of stacked choice-points. This allows chronological backtracking to an older state of computation in constant time by activating the corresponding environment. An additional advantage is that a separate *forward* slot is no longer necessary; all forwardings can be expressed through copy-pointers and environments. The unification algorithm itself proceeds roughly like standard destructive graph unification. Whenever two nodes are merged, a new copynode is only created when none of the two nodes is *active*, i.e. part of the current generation. When both nodes are active, they are destructively merged, if only one is active, the non-active node is forwarded to the active node.

The proposed algorithm is implemented in Common Lisp, and is used as the essential operation in the interpreter for the Typed Features Structure System TFS (Emele and Zajac,

1990a,b). The LIC method was evaluated on a sample HPSG grammar using a small set of test sentences, and an overall reduction in processing time of 60–70% is observed. Unfortunately, further details are not given.

One obvious disadvantage of Emele's method is the complex dereferencing operation, that will incur considerable overhead for a large grammar, where graphs are unified a large number of times to create a complex constituent. This is the same problem as in Pereira's algorithm, although the cost of chronological dereferencing should be small compared to application of environments as in Pereira's scheme. Another, more fundamental problem, that Emele's approach shares with any other incremental copying method, will be discussed in the next section. Strictly speaking, though, the tradeoff inherent in Emele's approach has never been evaluated and compared with other recent approaches (in particular that of Tomabechi, see next section); this remains an open question.

2.8. Tomabechi's Quasi-Destructive Graph Unification

The quasi-destructive unification introduced in Tomabechi (1991) and later detailed in Tomabechi (1995) is often considered the most efficient graph unification algorithm for natural language processing today (see e.g. van Lohuizen (2000)). Tomabechi's algorithm completely eliminates copying for unsuccessful unifications, and allows early finding of unification failures. Tomabechi starts out from the simple, yet important insight that unification does not always succeed[3], thus copying (which is known to be expensive) should only be done for successful unifications. Also, failures should be found as soon as possible. Tomabechi redefines the notion of early copying to include all copies created prior to finding a failure, while Wroblewski's definition only includes copies created before a failing unification starts. In retrospect, it seems artificial to exclude the copies made during unification in this way. All incremental copying methods necessarily suffer from early copying as defined by Tomabechi, since recursive calls into shared arcs cannot know if future recursions into other shared arcs will eventually fail, thus they cannot avoid to copy nodes.

The central idea in Tomabechi's unification algorithm is to provide a way to record changes made to a graph during unification in the nodes itself, without destructively modifying them. This is achieved by using an idea from Wroblewski (1987), namely that changes can be undone at virtually no cost when the changes are recorded in slots that are protected by a generation marker. The contents of such a temporary slot are ignored, unless their generation marker is equal to the global generation counter. All changes to a

[3] In fact, in typical large-scale unification-based parsing systems that use no filtering techniques, the vast majority of attempted unifications fails.

dag during unification are recorded in such temporary slots, i.e. in Tomabechi's scheme the *forward* slot is protected by a generation marker, and a new *comp-arcs* slot is introduced, where arcs added to a node during unification are recorded. When unification fails, all changes done so far can be undone by simply incrementing the global generation counter. When unification succeeds, the result is copied into a permanent structure, then the changes to the input graphs can again be undone by incrementing the global counter. The control structure of Tomabechi's algorithm is similar to Wroblewski's `unify1`, but all changes are made quasi-destructively. Tomabechi's algorithm can be seen as a way to completely delay copying, with only the minimal overhead of accessing temporary slots protected through a generation marker mechanism.

Tomabechi presents an evaluation of his algorithm, parsing 16 sentences using a HPSG-like grammar for Japanese that seems to be fairly small[4]. A speedup of up to a factor of two can be observed compared to Wroblewski's algorithm, and a similar reduction in the number of copies. I present a detailed evaluation on a large-scale grammar in Section 4.2.

2.9. Extensions to Tomabechi's Algorithm

While Tomabechi's unification algorithm, as presented in the previous section, completely eliminates early copying, the problem of redundant copying is ignored, i.e. the result graph consists of new nodes only, even if parts could be shared with the input graphs. Tomabechi (1992) and Malouf et al. (2000) present two similar ways to integrate structure sharing into Tomabechi's original algorithm.

The scheme suggested in Tomabechi (1992) follows two design principles, (i) atomic nodes can be shared, and (ii) complex nodes can be shared unless they are modified. No modification to the unification algorithm is necessary, structure sharing is achieved by the copying algorithm. The copying algorithm always shares (i.e. it does not copy) atomic nodes. Complex nodes are shared if no nodes below that node are changed; a node is considered changed if it is the target of forwarding, or if it has any temporary arcs. If a changed node is found in a recursion step, that information is passed up to the caller. Malouf et al. (2000) propose an algorithm that implements structure sharing "in much the same way" as Tomabechi, with a "crucial difference" concerning the sharing of nodes that are part of the grammar. Nodes that are part of the grammar must not be shared, since that could lead to spurious reentrancies or cyclic structures when one part of the grammar is used more than once in the derivation of a single sentence, or when graphs are modified by operations other than unification (e.g. a restrictor). An inefficient way to avoid the problem is instantiating the parse chart with fresh copies of all rules and lexical entries, and to copy structures before applying a restrictor. This extra copying can be avoided by marking

[4]The total grammar size is only 2324 nodes compared to several hundred thousand nodes in the large-scale grammars I am using, see Section 4.5.

nodes as *safe* or *unsafe* for sharing. All nodes that are part of the grammar are marked *unsafe*, nodes that are created during parsing are marked *safe*. The copy function checks this flag when deciding if a node can be shared. This improvement allows the parser to not copy a structure in many instances where it would have to be copied in Tomabechi's scheme. Both Tomabechi (1992) and Malouf et al. (2000) present an evaluation of their structure sharing method, and both report a speedup of about a factor of two, and an even bigger reduction in memory usage of up to a factor of three. I present an evaluation on the LinGO grammar in Section 4.2.

2.10. Van Lohuizen's Variant of Quasi-Destructive Graph Unification

Van Lohuizen (2000) presents a technique to reduce the memory usage of Tomabechi's unification algorithm (Tomabechi, 1991) considerably without increasing execution time, while making it thread-safe at the same time[5]. Tomabechi's algorithm uses temporary slots (*scratch fields*) in the node representation to avoid copying. These fields do not contribute to the definition of the graph, but are used for bookkeeping purposes in the unification and copying functions. Thus, they are not useful when a graph is not currently used in unification, but still fill up memory. Preferrably, scratch fields would be stored in a separate buffer that could then be reused. This has the additional advantage of increasing the probability that they remain in cache, which is an important consideration given the growing difference in speed between processor and memory. Separating the scratch fields from the node structure also allows concurrent unification, because each processor can work on a private scratch buffer. The problem to be solved is that of binding scratch structures to nodes. Van Lohuizen discusses two possible approaches, the straightforward one of using a hash table, and a novel one that is based on using an array of scratch structures and assigning unique indices to the nodes in a graph. The indices correspond to elements in the scratch structure array. Structure sharing can be implemented in this scheme in a way similar to Tomabechi (1992).

Van Lohuizen implemented the algorithm in Objective-C using a fixed arity graph representation, and presents an evaluation on a medium-sized grammar for Dutch for 22 sentences of varying length. This shows that the algorithm does not increase execution time compared to Tomabechi's original algorithm; there is even a 7% speedup that can be attributed to improved cache behavior. Memory utilization is reduced substantially by a factor of up to three for complex input. Van Lohuizen (2001) reports about the adaptation of the algorithm for the LinGO grammar. An evaluation using the *fuse* test set (see Section 4.1.2) indicates a minimal decrease (1.7%) in performance compared to

[5] The technique is also applicable to Wroblewski's `unify2` (Wroblewski, 1987).

Tomabechi's algorithm, and a significant reduction of memory utilization by a factor of almost two. Another experiment using concurrent unification shows a remarkable speedup of nearly two on a dual processor machine for the *fuse* test set.

2.11. Summary

The papers I discussed in the previous sections provide an interesting example of scientific evolution. Techniques that turned out to be of essential importance can be found very early, but their potential was often realized only much later, partly because their effectiveness in isolation was limited and required synthesis with other techniques, and partly because they were only vaguely documented, and not backed by empirical evaluation. One example is the idea of structure sharing, that dominated the discussion in the beginning (Karttunen and Kay, 1985; Pereira, 1985), but then was ignored for quite a while, and only picked up again by Kogure (1990). The full potential of the generation counter mechanism from Wroblewski (1987) was only realized in Tomabechi (1991). Another main ingredient of Tomabechi (1991) is already presented in Karttunen (1986), namely to only copy when unification is successful, by making unification reversible. Karttunen only briefly discusses this idea in his report, and provides no details of the implementation, nor an evaluation. Also, only the combination with the generation counter mechanism allowed undoing unifications cheaply. A final example is the idea of using failure-frequency information to find failing unification as early as possible, that is first documented in the *SING* method from Kogure (1990), and generalized in Uszkoreit (1991). In both approaches failure-frequency information is used to guide the unification algorithm. Erbach (1991b) is the first to describe a filter that allows to skip unifications that if executed would fail; a similar filter is later independently proposed by Maeda et al. (1994). These methods, however, are based on manually compiled static information about failure-points in feature-structures. The technique that combines the ideas of filtering and using failure frequency information is the *quick check* (Malouf et al., 2000; Kiefer et al., 1999), which I will discuss in Section 4.4. The algorithm presented in Tomabechi (1991) with the structure-sharing improvements from Tomabechi (1992) and Malouf et al. (2000) can be seen as the successful synthesis of the previous approaches, avoiding all kinds of unnecessary copying as far as possible. This will be the main algorithm used in my empirical work in Chapter 4. Van Lohuizen (2000) is a further improvement, significantly reducing the memory requirements. A common trend in the papers can be noted. While the early papers often provide no empirical evaluation at all, all the recent ones include detailed empirical results, with increasingly challenging input used over time. A similar observation can be made for the level of detail in that the algorithms are presented. While the early papers often only verbally sketch the algorithm, detailed pseudo-code is included in most recent publications. Both facts greatly increase the ability to reproduce and compare the published results.

There is universal agreement in the discussed literature that copying is the main source of inefficiency in graph unification. Over copying, early copying and redundant copying are accepted measures to quantify the different kinds of unnecessary copying. Nevertheless, no precise evaluation has taken place, quantifying the amount of the different kinds of copying. Section 4.2 reports about an experiment that answers this question. It is often claimed that unification (and the necessitated copying) take the vast majority of run time in a unification-based processing system (Karttunen and Kay, 1985; Godden, 1990; Kogure, 1990; Tomabechi, 1991, 1992; van Lohuizen, 2000; Malouf et al., 2000), but it is an open question what the actual distribution of run time in a system combining the techniques to reduce unnecessary copying and filtering techniques looks like. I discuss this in Section 4.3. The idea of representing feature structures using a skeleton/environment scheme from Pereira (1985) was not pursued in later work. I discuss an application of this representation in Section 4.7, and compare it with two alternative strategies.

2.12. Alternative Approaches

While there is a significant body of relevant work besides what I have discussed in the previous sections, mainly from the area of logic programming and the implementation of logic programming languages, a full discussion would be outside the scope of this thesis. An easily accesible overview over some of the progress in this area can be found in (Van Roy, 1994). This section briefly surveys alternatives to the implementation of unification as graph unification. The alternatives can broadly be classified into two categories. The first category comprises approaches based on compilation to customized abstract-machines, most of them modelled after the *Warren Abstract Machine* (WAM) (Warren, 1983a; Aït-Kaci, 1991).

$\mathcal{AMALIA}$ (Wintner and Francez, 1995; Wintner, 1997; Wintner et al., 1997; Wintner and Francez, 1999) and the machine proposed in Qu (1994); Carpenter and Qu (1995) are two examples. LiLFeS (Makino et al., 1997, 1998; Miyao et al., 2000; Torisawa et al., 2000) can be considered an implementation of the latter, combining it with techniques from Aquarius Prolog (see Penn, 2000, Chapter 8). CHIC (Ciortuz, 2000), or *Light* as it has recently been renamed to, is another approach using direct compilation, this time based on the abstract machine architecture proposed in Aït-Kaci and Di Cosmo (1993), that is grounded in the theoretical foundation of order-sorted feature unification (Aït-Kaci et al., 1993). Brown and Manandhar (1998) and Brown and Manandhar (2000) discuss another compilation-based approach, which is based on precompilation of all possible feature structures generated during parsing[6].

[6] Although, somewhat misleadingly, labeled an abstract-machine-based approach, this work could better be compared with approaches using context-free approximation; see Kiefer and Krieger (2000).

Given that all these approches are based on compilation, it is interesting to look at the motivation that Carpenter and Qu (1995) provide for compilation:

> *The vast majority of the time and space used by traditional unification-based grammar interpreters is spent on copying and unifying feature structures. [...] The principal drawback to this approach is that complete feature structures have to be constructed, even though unification may result in failure. [...] By adopting an incremental compiled approach, a description is compiled into a set of abstract machine instructions. At run-time a description is evaluated incrementally, one instruction at a time. In this way, conflicts can be detected as early as possible, before any irrelevant structure has been introduced.* (Carpenter and Qu, 1995, p. 1)

In the light of the results from the previous sections, this is exactly what the algorithm proposed in Tomabechi (1991) achieves for interpreted unification.

The second category comprises approaches that map unification of feature structures to a logic programming language (most commonly Prolog). ALE (Carpenter, 1992a; Carpenter and Penn, 1996) and ProFIT (Erbach, 1994) are prominent examples of this approach. The idea of falling back on a stable and thoroughly optimized language like Prolog is tempting, although ProFIT and classic ALE could not rival direct approaches in performance. However, Penn (2000) introduces a new optimal flat-term encoding, that significantly improves upon tree encodings (as used in ProFIT) in terms of both speed and the range of grammars that are covered, and hence convincingly argues for this approach. Penn concludes that in the light of his results the value of building customized abstract-machines is questionable:

> *There is, in fact, a small cottage industry of abstract machines for feature-structure-based natural language processing now [...], abetted by careless, inaccurate benchmarking that exaggerated their improvement relative to [...] systems such as ALE and ProFIT. That includes, for example, ignoring that different parsing algorithms and/or chart-indexing strategies were used, using very small test corpora (often fewer than 10 sentences) and using test sentences of such very small complexities that the initialization routines are more computationally significant than the parsing routine itself.*
> (Penn, 2000, p. 200)

Penn then compares the performance of LiLFeS 0.88 compact code and his own encoding in ALE on the HPSG grammar distributed with ALE, using identical parsing algorithms. At the closest separation, ALE running on SICStus is 2.5 times faster, with LiLFeS's performance slowly degrading to slightly over 10 times slower for more complex input. Penn concludes:

> *LiLFeS's slower performance is mostly due to the fact that SICStus's memory management and predicate compilation are simply much better. On the other hand, this is one of the main reasons for using a Prolog-based implementation to begin with: avoiding redundant problem-solving and utilizing the last sixteen years' worth of research on optimizing the WAM.*
>
> *(Penn, 2000, pp. 201f.)*

Gerald Penn is working on an evaluation of his encoding using the LinGO grammar. It will be interesting to see if the results from above carry over to large-scale grammars. Currently, the unifier performance reported in Miyao et al. (2000) (in terms of unifications that can be executed per second when no filtering is used) represents best practice for the LinGO grammar; on the other hand, it is not clear if the techniques that were highly successful in improving the speed of the parsers in interpreted systems, like quick check filtering or key-driven parsing, can be integrated into a system like LiLFeS with comparable benefits.

3. The PET System

This chapter discusses the PET platform I implemented to serve as a flexible basis for the empirical study that is the topic of Chapter 4, allowing precise empirical study and comparison of different approaches to efficient processing of large-scale unification-based grammars. A second, orthogonal goal was pursued, namely to provide an efficient run-time processing system that allows fruitful scientific and practical utilization of HPSG grammars, complementing the existing development platforms PAGE and LKB. PET is freely available under an open-source licence, see Appendix D. The following sections set out with an overview of PET, then discuss the preprocessor and its efficient implementation of semi-lattice computation, and conclude with a discussion of the run-time system.

3.1. Overview

PET is a platform for experimentation with processing techniques and the implementation of efficient processors for unification-based grammars. It synthesizes a range of processing techniques from earlier systems into a modular C++ implementation, supplying building blocks from which experimental setups can be configured. This allows the precise empirical study of practical performance and the contrastive comparison of different approaches and their interaction in a common context. Underlying PET is the common descriptive formalism (see Chapter 1) that is also implemented in the LKB system (Copestake, 1992, 1999). This makes it possible to process the three large-scale HPSG grammars of English, Japanese, and German available in this formalism with PET, and to use them for my empirical work (Chapter 4). Also, a parser built from PET components can be used as a time- and memory-efficient run-time system for grammars developed in the LKB. In daily grammar development it facilitate frequent, rapid regression tests.

PET consists of two main parts, a preprocessor called flop (discussed in Section 3.2), and the run-time system (typically a parser configured from PET components). A chart parser called cheap is the preconfigured standard parser for PET, it is the topic of Section 3.3. Figure 3.1 gives an overview of the system and the experimental setup.

The flexibility and extendibility required for the kind of empirical study I propose is achieved by a tool box approach: PET provides an extendible set of configurable building

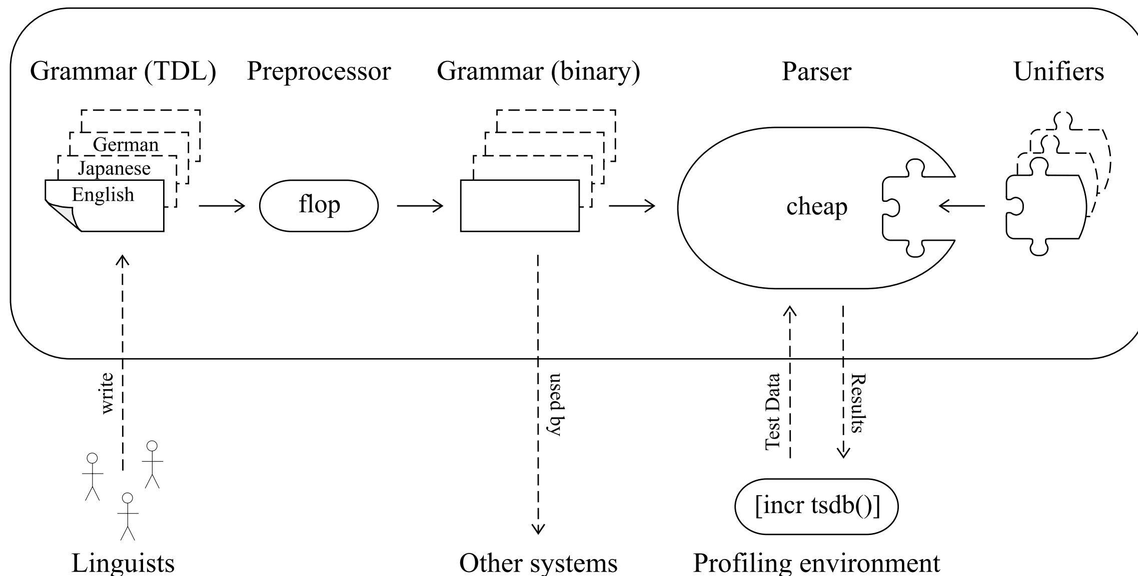

Figure 3.1.: PET – System Overview and Experimental Setup

blocks that can be combined and configured in different ways to instantiate a concrete processing system. The set of building blocks includes objects like *chart*, *agenda*, *grammar*, *type hierarchy* and *typed feature structure*. For instance, a simple bottom-up chart parser can be implemented using the available objects in a few lines of code. Using alternative implementations of one object allows controlled experiments, comparing different approaches to one aspect of processing, leaving everything else unchanged. For instance, the *typed feature structure* object can be configured to use a number of different unification algorithms and feature structure representations, including `unify1` and `unify2` from Wroblewski (1987), `unify3` from Section 2.4.2, the algorithm from Tomabechi (1991) with optional structure-sharing from Malouf et al. (2000), and finally three variants of Tomabechi's algorithm using fixed-arity-based feature structure encodings (these are discussed in detail in Section 4.6). My goal was to implement the most influential algorithms for graph unification. Wroblewski's algorithm was chosen because it is the root of all algorithms proposed later, and serves as a common baseline. Tomabechi's algorithm (plus structure sharing) was selected because it can be considered the synthesis of previously discussed approaches (see Section 2.11). I did not include the algorithm from van Lohuizen (2000), simply because it was not published when I implemented PET (an evaluation of the algorithm on LinGO is documented in van Lohuizen, 2001).

PET is implemented in ANSI C++, and compiles on a wide range of Unix machines using the GNU C++ compiler; for Windows platforms it can be compiled using Borland C++, or the Cygnus port of GNU C++ (cygwin). The full system comprises roughly

25,000 lines of code, of which 5,500 lines are shared between the preprocessor and the run-time system, the preprocessor itself is 7,100 lines, and the run-time system comprises 12,200 lines. Special attention was paid to efficiency and compactness when developing PET. Critical objects are carefully optimized. PET uses traditional C representations (rather than C++ objects) in places where minimal overhead is required, e.g. for the basic elements of feature structures. Information about the usage of PET is given in Appendix C.

3.2. The Preprocessor

A number of tasks like syntax and consistency checking for the source grammar, expansion of constraints and conversion into a representation suitable for efficient processing have to be repeated each time a grammar is used for processing. To achieve minimal startup time for experimentation (where often a number of experiments is made using an identical grammar) and production use, these tasks were factored out in the PET system and moved into a preprocessor. The preprocessor (called flop) reads the source form of a grammar and converts it into a binary representation that is read by the run-time system. The preprocessed form is also a convenient entry point for other systems[1], avoiding duplication of effort in the implementation of preprocessing techniques. Appendix B specifies the output file format.

3.2.1. Overview

The preprocessor reads a grammar in a variant of $\mathcal{TDL}$ (Krieger and Schäfer, 1994a) (defined in Appendix A), expands $\mathcal{TDL}$ templates, constructs a lower semi-lattice from the type hierarchy (see Section 3.2.2), infers appropriateness conditions, performs configurable expansion (Krieger and Schäfer, 1995) and unfilling (Götz, 1993; Gerdemann, 1995) of type definitions (see Section 4.5), and optionally decomposes the type-hierarchy for feature structure encoding purposes (see Section 4.6.4). The result of preprocessing is a compact, binary representation of the grammar (defined in Appendix B) that is designed for efficient loading by a run-time system.

The source of the LinGO grammar, about 52,000 lines of $\mathcal{TDL}$, is preprocessed on a 500 MHz Pentium III in 11.5 s, resulting in a 2,260 kb output file that the run-time system loads in less than one second; the Japanese grammar, about 34,000 lines of $\mathcal{TDL}$, takes 3.9 s for preprocessing, resulting in 1,635 kb of output. The shorter preprocessing time mainly results from the smaller number of types in the Japanese grammar (only 3693

[1]For example, the cali system (van Lohuizen, 2001) is using grammars preprocessed by flop; this decision saved considerable development time, as idiosyncrasies of the grammar source format are completely circumvented.

types (of which 1208 are non-leaf types[2]) compared to 7188 (with 1636 non-leaf types) in the LinGO grammar). The German grammar, about 139,000 lines of $\mathcal{TDL}$ source, is preprocessed in 53.8 s, resulting in a 10,922 kb output file. The long preprocessing time can be attributed to the larger number of non-leaf types, which makes semi-lattice computation less efficient (the grammar has 3564 types, of which 2723 are non-leaf types), and the large lexicon (220,197 full forms, compared to 17,917 in LinGO and 3,654 for the Japanese grammar). For comparison, preprocessing and loading a grammar cannot be separated in PAGE and LKB, where the whole process of reading the LinGO grammar takes 96.1 s and 48.5 s[3], respectively.

3.2.2. Semi-lattice Computation

The formalism assumed in PET requires that the type hierarchy is a finite bounded complete partial order (*BCPO*), i.e. every consistent set of types must have a unique greatest lower bound. It is, however, not necessary to require the grammar writer to specify only type hierarchies that satisfy this condition; instead, a *BCPO* embedding the specified hierarchy can be constructed automatically by adding types to the hierarchy. The underlying theoretical construction is discussed in Aït-Kaci et al. (1989, section 3); some clarifications are found in Penn (2000, section 2.1.2), where a polynomial bound on the number of added types is proven. This section discusses an efficient implementation[4] of this construction.

In the PET preprocessor, the type hierarchy is represented as a directed graph, with nodes in the graph corresponding to types, and edges corresponding to immediate subsumption. This representation corresponds directly to the way the type hierarchy is specified by the grammar writer, and it provides convenient access to information about immediate ancestors and descendents of a given type, which is the most commonly requested information during preprocessing.

For semi-lattice computation I make use of the basic transitive reflexive closure encoding from Aït-Kaci et al. (1989, section 4).[5] I use a hash table to compute the bit code to type correspondence. The bit codes are assigned analogous to the `AssignCode` procedure from Aït-Kaci et al. (1989, section 4.2). This guarantees that there is a straightforward perfect hash function for the bit codes, namely the function that returns the position of the first non-zero bit in the code. It is a perfect hash function in the sense that for any code in the table it returns a unique value (a proof follows easily from the construction), it will not

[2] I call types that have no descendants, and only a single immediate ancestor *leaf types*.

[3] This is reduced to 9.7 s when using a cached lexicon.

[4] Semi-lattice computation is the single most time consuming task in preprocessing in my implementation, which makes efficiency an interesting concern.

[5] This encoding is also used in the run-time system, with an additional cache for the computation of type intersection, see Section 3.3.

return unique values for every possible bit code, obviously. Since each bit code has n bits, where n is the number of types, this function always yields a value in the range $[0, n]$.

The input to the *BCPO* construction algorithm is the type hierarchy as specified in the grammar, represented as a graph; the output is a graph representation of the embedding *BCPO*. The computation consists of three main steps[6]:

1. We compute the bit code $\gamma(t)$ for each type t, using an iterative version of the `AssignCode` function from Aït-Kaci et al. (1989, section 4.2).

2. We consider each (ordered) pair (t_1, t_2) of types, and compute c as the bitwise *and* of $\gamma(t_1)$ and $\gamma(t_2)$. If c is not null, and not the code of an existing type, this indicates t_1 and t_2 do not have a unique greatest lower bound. For any such pair we create a new type t_3 with $\gamma(t_3) = c$. This is iterated until no more types are added. The performance of this phase crucially depends on fast lookup of existing bit codes, and thus benefits greatly from the perfect hash function.

3. Now we reconstruct the graph representation of the type hierarchy from the bit code representation in two steps:

   a) The first step creates the graph, possibly containing redundant links (for non-immediate subsumption). The naïve approach ignores the existing input graph, and looks at all pairs of types, checks for subsumption using their bit codes, and adds an edge for all pairs of types that are in a subsumption relation. We can do better by starting from the original input graph. Then we consider each new type t. We efficiently enumerate all sub-types of t by looking at $\gamma(t)$ (sub-types of t correspond to the bits in $\gamma(t)$ with a value of 1) and add the corresponding edges to the graph. We also add edges for all types from the original hierarchy that are super-types of t, by iterating over all original types and checking for subsumption using the bit code.

   b) Removing redundant links is achieved by computing the transitive reduction of the graph, using a standard algorithm (Mehlhorn, 1984).

   This phase depends heavily on the efficiency of the bit vector operations that are used in the implementation of the subsumption test for bit codes.

There is a simple yet effective optimization to the algorithm described above. In real-world hierarchies there often is a large number[7] of types that have no descendants, and only

[6] If you are interested in more details (or the actual implementation), please refer to the (liberally commented) file `flop/hierarchy.cpp` in the sources (see Appendix D).

[7] More than two thirds of all types for both LinGO and the Japanese grammar, but only about one quarter of all types for the German grammar.

a single immediate ancestor, so-called leaf types in the LKB system. In PET these types are automatically identified and excluded from the semi-lattice computation. Another possible optimization is to partition the types into independent blocks before semi-lattice computation. When the *BCPO* construction algorithm of PET was recently imported into the LKB by John Carroll, this optimization was added. For the LinGO grammar it results in a significant speedup of about a factor of 20.

3.3. The Parser

The preconfigured run-time processor in PET is a chart parser (Kay, 1986) called cheap. The basic parsing algorithm is a variant of the bottom-up parsing algorithm from Erbach (1991a), that operates bidirectionally in instantiating rules and processing the input string. cheap can be run in exhaustive (all paths) or agenda-driven best first search modes. As motivated in Chapter 1, cheap is integrated with the [incr tsdb()] competence & performance profiling environment. Section 4.1 provides details about the role of [incr tsdb()] in my work.

3.3.1. Techniques Applied for Efficient Processing

cheap employs all relevant techniques for efficient processing from Kiefer et al. (1999) as well as other techniques originally developed in PAGE and the LKB. Reimplementing techniques imported from other systems often allowed for improved engineering, because previous experience was available, and specific requirements could be accounted for in the design phase. An example of this is the quick check, where cheap uses a tree-like structure rather than a list of paths, as used in the LKB or PAGE (for details, see Section 4.4).

Type operations For efficiency reasons, types are represented as integer numbers at run time. The preprocessor assigns a unique number from the range $[0 \ldots n_{\text{types}}[$ to each type. This range is further divided into four disjunct, consecutive ranges:

$$\underbrace{[0 \ldots n_1[}_{\text{proper types}} \; \underbrace{[n_1 \ldots n_2[}_{\text{leaf types}} \; \underbrace{[n_2 \ldots n_3[}_{\text{symbols}} \; \underbrace{[n_3 \ldots n_{\text{types}}[}_{\text{instances}}$$

The concept of leaf types was explained in Section 3.2.2, symbols are all the subtypes of the built-in special type atom[8], and instances are a relict initially implemented for PAGE-compatibility that is now used to represent lexical entries. The difference to leaf types is that feature structures of instances are only expanded on demand. The remaining types

[8] All string constants in a grammar are subtypes of atom, so there usually is a flat, but extremely large subhierarchy below atom. This justifies the special treatment of these types.

are called proper types. A division like this was chosen because types from some of these categories can be handled more efficient than the general case. Full greatest lower bound (*glb*) computation, for example, only has to be done for proper types; for leaf types, symbols and instances it is reduced to glb computation on their parent types. Consecutive ranges allow cheap tests (integer comparisons) to determine what category a given type belongs to.

Unlike suggested in Kiefer et al. (1999) I do not use a full precomputed table of glbs, but rather cache type intersection at run time in a hash table. In agreement with Malouf et al. (2000) I found this to be as efficient in terms of time, while greatly reducing memory consumption. Since glb computation still takes a significant percentage of total runtime (see Section 4.3), I compared a hash table implementation from the standard library with a streamlined custom implementation, and found the latter to increase total parser performance by nearly 10%. My implementation only supports those operations on a hash table necessary to implement the glb cache, i.e. construction, a combined search and update function (`operator[]`), and destruction. Deletion of elements from the table, for instance, is not supported. I use hashing with chaining (for information on hashing and the notation I use here, see Cormen et al., 1990, Ch. 12) and the simple hash function $h(k) = k \bmod m$, where k is the key, and n is the size of the hash table, which is always chosen as a prime number. For a pair (t_1, t_2) of types[9], the key is computed as $k = t_1 n_{\text{types}} + t_2$. This is a unique key, since n_{types} is greater than any t. I empirically determined (using the *fuse* test set on LinGO) a suitable size for the hash table that balances access time and space requirements. Interestingly, I found using only 12,289 buckets to be a good compromise, even though the hash table contains a total of 33,696 entries after parsing the *fuse* test set, yielding a load factor $\alpha = 2.74$. Still, in slightly more than 93% of the total 705,689,857 searches the element is immediately found in the table; only for 7% of all searches the element is found in overflow buckets. Even then, the average length of a chain of overflow buckets that has to be followed is only 1.15. This explains the excellent performance of the hash table. My implementation is encapsulated in a class that can be fully inlined by the compiler, further improving performance. For the actual implementation, please refer to `common/glbcache.h` in the sources.

Key-driven parsing cheap (optionally) uses a bidirectional, key-driven parsing strategy (Kiefer et al., 2000; Oepen and Callmeier, 2000), originally implemented in PAGE. The key-driven parsing strategy avoids proliferation of active items in the parser[10] for rules that contain very unspecific argument positions, by first instantiating the argument position

[9]Since type intersection is commutative, I reduce type intersection of (t_1, t_2), where $t_2 < t_1$, to the intersection of (t_2, t_1). This halves the potential number of pairs to consider.

[10]A complementary solution for the problem of proliferation of active items is described in Erbach (1991b), where feature structures of active items are not preserved to save memory. This is similar to hyper-active parsing, see below.

that best constrains the rule's applicability. For terminological clarity, PAGE introduces the term *key* daughter for this argument position of a rule. While head-driven approaches to parsing have been explored successfully with lexicalized grammars like HPSG (for an overview, see van Noord, 1997), many authors (Kay, 1989; Bouma and van Noord, 1993) assume the *linguistic head* to be the argument position that the parser should instantiate first. The notion of *key-driven* parsing emphasizes the observation that for individual rules in a particular grammar the non-head daughter may be the better candidate. For key-driven parsing in cheap, the grammar writer specifies the key-daughter for each rule, either as an annotation in the rule, or in the `flop.settings` configuration file (see Appendix C). The right choice of key-daughter in each rule, such that it best constrains rule applicability can hardly be determined analytically, but for a given test set and given grammar with at most binary-branching rules, say, it can be determined by parsing the test set twice, once using strict left to right and once using strict right to left rule instantiation, and then comparing the number of active items postulated for each rule between the two runs. The number of test runs, of course, increases with the branching factor of the grammar. The required statistics can be compiled by cheap (using the option `-rule-stats`) and then evaluated using [incr tsdb()] (see Oepen and Callmeier, 2000).

Restriction A restrictor (Shieber, 1985) to be applied to passive items can be specified by the grammar writer. This allows to apply the technique from Kiefer et al. (1999) that avoids duplication of the derivational structure in feature structures of passive items during parsing, by removing the daughter structure from the feature structures of passive items.

Filtering cheap implements the two pre-unification filters from Kiefer et al. (1999), the (static) rule filter, and the (dynamic) unification filter called quick check. The first filter avoids execution of failing unifications by referring to a table that specifies for a pair of rules and an argument position if the second rule can be unified into the given argument position of the first rule. The second filter is discussed in Section 4.4.

Hyper-active parsing cheap (optionally) employs the hyper-active parsing strategy presented in Oepen and Carroll (2000b); Oepen and Callmeier (2000) and originally implemented in the LKB. In hyper-active parsing, when an active edge is derived, the partial analysis is stored in the chart, but the associated feature structure is not copied; it is, however, used to compute the information required for the quick check filtering method. When the active edge is actually combined with a passive edge, the intermediate feature structure is recomputed from the original rule and daughters. The feature structures of (complete) passive edges are copied as usual. Storing active edges without expensive feature structure copying enables the parser to perform a key-driven search effectively, and at the same time avoids over-copying for partial analyses; additional unifications are traded for the copies

that were avoided only where hyper-active edges are actually extended in later processing. An additional optimization (termed *excursion* in Oepen and Carroll, 2000b) is applied when the unification algorithm from Tomabechi (1991) is used. The parser is allowed to deviate from the agenda-driven control strategy to try combination of the active edge with one suitable passive edge while the feature structure of the active edge is still valid (i.e. within the same unification generation). Hyper-active parsing can be disabled for all rules by using the `-no-hyper` option to cheap, and selectively for certain rules by specifying them in the `cheap.settings` configuration file[11]. Please see Section 4.7 for the discussion of an alternative to redoing unifications in order to recreate the feature structure of an active item in hyper-active parsing.

Limiting the number of initial chart items cheap supports the specification of mutual dependencies of certain lexical items, and can (optionally) remove items that do not satisfy these dependencies from the chart after chart initialization, as described in Kiefer et al. (1999). Dependencies are specified using the option `chart-dependencies` in the `cheap.settings` configuration file. Currently, this feature is used only in the German grammar.

Ambiguity packing Initial support is provided for ambiguity packing as described by (Oepen and Carroll, 2000a) and implemented in the LKB. The parser can be configured (using the compile-time option `-DPACKING`) to produce a packed chart; no support, however, for unpacking the chart, or selecting readings from it is currently provided.

Summary The combination of these techniques results in attractive performance for the cheap parser, both in terms of speed and memory consumption. All the twenty-word sentences from the *fuse* test set (see Section 4.1.2) for the LinGO grammar can be parsed exhaustively (using an upper limit on the chart size of 20,000 passive edges) in a process size of 78 Mb[12], in an average time of 1.7 s per sentence.

3.3.2. Memory Management

Efficient memory management and minimizing memory consumption was an important consideration in the implementation. Experience with Lisp-based systems has shown that memory management is one of the main bottlenecks when processing large-scale grammars. In fact, one observes a close correlation between the amount of dynamically allocated memory and processing time, indicating much time is spent moving data, rather than in actual computation. Performance profiles of an early version of cheap that used

[11] See Oepen and Callmeier (2000) for why this could be useful.

[12] Using the minimal fixed arity encoding presented in Section 4.6 reduces this to 60 Mb.

the built-in C++ memory management supported this. Allocation and release of feature structure nodes was accounting for almost 40% of the total run time. However, like in the WAM (Warren, 1983a; Aït-Kaci, 1991), a general memory allocation scheme allowing arbitrary order of allocation and release of structures is not necessary in this context. When parsing we typically continue to build up structures. Memory is only released in the case of a top-level unification failure when all partial structures built during this unification are released. Therefore, cheap uses a simple and efficient stack-based memory management strategy provided by PET, where memory is acquired from the operating system in large chunks and then sub-allocated. There is no way to release individual objects; instead a *mark-release*-mechanism allows saving the current allocation state (the current stack position) and returning to that saved state at a later point. Thus, releasing a chunk of objects amounts to a single pointer assignment. Switching to this memory management implementation resulted in a significant overall speedup (a little less than a factor of 1.6) for the early version of cheap mentioned above.

4. Empirical Results

Empirical study is indispensable for the evaluation and optimization of the practical performance of constraint-based processing systems. As Carroll (1994) argues, we do not yet have the analytic tools that would allow us to predict how the properties of individual unification-based grammars will interact with particular processing techniques. To obtain meaningful results, empirical study requires controlled experiments on large sets of data. For instance, the practical performance of a unification algorithm depends on a large number of factors, ranging from the more obvious, like the complexity of the input data, the parsing strategy employed, the underlying type system, and the implementation programming language, to less obvious factors like individual programming style. Thus, a meaningful empirical comparison of processing techniques can only be made if the techniques are evaluated in a common context. Obviously, the interaction between several techniques can *only* be studied when they are implemented in a common system. To abstract from peculiarities of the input, an evaluation using several grammars and large sets of input data is highly desirable.

This chapter presents the results of a series of experiments using the PET platform introduced in the last chapter. The collection of experiments as reported here was chosen in order to answer the most important questions raised in Chapters 2 and 3, and at the same time to illustrate the wide range of questions that can be empirically answered using a flexible experimentation platform like PET in conjunction with the [incr tsdb()] profiling environment. The first section discusses the common experimental setup, and summarizes relevant details of the grammars and test sets I use. In Section 4.2, I take a closer look at the cost of copying for different unification algorithms, quantifying the amount of early, over and redundant copying. The discussion includes a comparison of the performance of Wroblewski's and Tomabechi's unification algorithms. Section 4.3 discusses the distribution of run time in the unifier, using execution profiles obtained with a Unix execution time profiler (`gprof`). The next section is concerned with the quick check pre-unification filter. In Section 4.5, I look into the benefits of two techniques for reducing the size of feature structures at run time, called unexpansion and unfilling. Section 4.6 moves on to a survey of different feature structure encoding techniques, and empirically compares lists of feature-value pairs with fixed-arity-based representations. I discuss three different techniques for an active chart parser to deal with the feature structures associated to active

Measure	Description
i-length	length of test item in words (see Oepen et al., 1997)
words	number of lexical entries retrieved
readings	number of complete analyses obtained
pedges	number of passive edges built (typically in all-paths search)
filter	percentage of parser actions predicted to fail by filters
etasks	number of attempts to instantiate an argument position in a rule
stasks	number of successful instantiations of argument positions in rules
unifications	number of top-level calls into the feature structure unification routine
copies	number of top-level feature structure copies made
tcpu	amount of cpu time spent in processing
space	maximum amount[a] of memory allocated at any point during processing
dspace	total amount[b] of memory allocated during processing
fssize	average number of nodes[c] in feature structures of passive edges

[a]Note that this deviates from the original definition in Oepen and Callmeier (2000). The new definition provides a direct handle on the amount of memory required by the parser.

[b]This corresponds to the original definition of *space*.

[c]Nodes that are the destination of a reentrancy are counted once for each reference.

Table 4.1.: Summary of relevant profiling parameters (adapted from Oepen and Callmeier, 2000).

edges, namely copying, recomputation, and trailing in Section 4.7. Finally, Section 4.8 concludes with a discussion of how to predict practical performance across processing platforms and grammars.

4.1. Experimental Setup

An overview of the experimental setup was shown in Figure 3.1 on page 26 in the previous chapter. This section discusses the role of the [incr tsdb()] environment, contains a compilation of the measures I use, and reviews relevant properties of the grammars and test sets. Finally, technical details of the experimental environment are provided. The raw results of all experiments are available online, see Appendix D.

4.1.1. The [incr tsdb()] Environment

I use the [incr tsdb()] *competence & performance profiling* environment (Oepen, 2001; Oepen and Carroll, 2000b; Oepen and Callmeier, 2000; Oepen and Flickinger, 1998) for my experiments. [incr tsdb()] provides facilities to obtain, analyze, and compare rich,

precise, and structured snapshots of system behavior (called *profiles*). Except for the experiments in Sections 4.3 and 4.4, I collected all empirical data using the graphical [incr tsdb()] user interface. I compared all results (number of readings obtained, number of passive edges, and the actual derivation trees) with reference results obtained in the LKB system to ensure comparability (and correctness with respect to the LKB as a reference system). For all data on the LinGO grammar this yielded an exact match; for the Japanese and German grammar some differences could be observed. They could, however, all be attributed to differences in the lexicon[1]. For these two grammars, when comparing performance between systems, I restricted the test sets to items where identical results were obtained. An upper limit of 20,000 edges on the number of passive edges in the chart during parsing was imposed for all experiments.

Measures

The [incr tsdb()] environment defines a common set of descriptive metrics which aim for in-depth precision and also for sufficient generality across processing systems. An [incr tsdb()] profile consists of information on

> *(i) the processing* environment *(grammar, platform, versions, parameter settings and others), (ii) grammatical* coverage *(number of analyses, derivation and parse trees per reading, corresponding semantics), (iii)* ambiguity *measures (lexical items retrieved, number of active and passive edges, where applicable, both globally and per result), (iv)* resource consumption *(various timings, memory allocation), and indicators of (v)* parser *and* unifier *throughput.* *(Oepen and Callmeier, 2000)*

The discussion in the remainder of the chapter is mainly concerned with measures from (iv) and (v) above. Table 4.1 summarizes the profiling parameters relevant to the discussion in this chapter.

The common metric has greatly increased comparability and data exchange among different groups in our collaboration (see Chapter 1), and has in some cases also helped to identify unexpected sources of performance variation. For example, we have found that two Sun UltraSparc servers (at different sites) with identical hardware configuration (down to the level of cpu revision) and OS release reproducibly exhibit a performance difference of around ten per cent. This appears to be caused by different installed sets of vendor-supplied operating system patches.

[1] The LKB analyzes morphology online, while PET is using a full-form lexicon.

test set	# of items	i-length	words	readings	pedges
aged	96	8.41	27.77	16.29	526
fuse	2,161	11.62	42.90	69.55	1,850
vm-1	2,838	8.52	25.96	104.80	725
vm-1$_{\geq 15}$	303	17.94	50.72	527.61	3,326
balance	1,464	8.18	37.46	4.62	3,355

Table 4.2.: Properties of the test sets used for the experiments. All test sets are restricted to those items that can be parsed exhaustively using the imposed limit of 20,000 passive edges in the chart.

4.1.2. Grammars and Test Sets

My experimental work is based on three large-scale HPSG grammars, the LinGO grammar of English (Flickinger et al., 2000a; Copestake and Flickinger, 2000), the Verb*mobil* grammar of Japanese (Siegel, 2000), and the Verb*mobil* grammar of German (Müller, 1999; Müller and Kasper, 2000). All three grammars obey a common descriptive formalism (see Chapter 1) and can be processed by PAGE, LKB, and PET[2]. The LinGO grammar was the first grammar among the three that was available in the common formalism; in fact, it served as a reference point for this formalism for quite some time[3]. I am using the reference version of LinGO frozen for Oepen et al. (2001), except in Section 4.4, which is based on the version frozen for Flickinger et al. (2000b). The (somewhat smaller) Japanese grammar is available in the common formalism since early 2000. I am using the May 2000 version for all experiments. The German grammar is the most recent member in this collection of grammars; my experiments are based on the January 2001 version.

The test sets are all based on data from Verb*mobil*. For English, I am using two different test sets[4], a small test set of 96 utterances, the so-called *aged* test set, and a larger set of 2,363 items, called *fuse*, which is randomly extracted from Verb*mobil* corpora so that a balanced distribution of 100 samples for each input length below twenty words was achieved. The *fuse* test set was compiled for Oepen et al. (2001) by Dan Flickinger and Stephan Oepen. For Japanese, I am using a collection of 2,838 test items from Verb*mobil*, compiled by Melanie Siegel and called *vm-1*. Since this test set is a lot less demanding

[2]This is not an exhaustive list, there are a number of systems (e.g. CHIC, LiLFeS and cali) that have been shown to process LinGO; they should, in principle, be able to process the other two grammars.

[3]Given that during the initial PET development LinGO was the only available grammar, a certain amount of tuning to the specific requirements of LinGO cannot be ruled out; it is certainly the grammar whose behavior is most extensively studied, both in PET and the LKB.

[4]Actually, Section 4.4 makes uses of another (third) test set, very similar in design to *fuse*, but slightly smaller. It is called *blend* and was compiled for Flickinger et al. (2000b) by Dan Flickinger and Stephan Oepen.

than the test sets for the other grammars, I restricted this set to all items of a length of 15 words or more to ease comparison with results on the other grammars; I labelled the the restricted, more demanding set *vm-1*$_{\geq 15}$. For German, I extracted a sample of 2,000 test items, containing 100 samples for each input length between 1 and 20 words, out of Stefan Müller's collection of Verb*mobil*-corpora. The main properties of these test sets relevant to the discussion in this chapter are summarized in Table 4.2. Only *i-length* is a property of the test data itself, the indicators for average ambiguity were obtained with PET on the grammar versions mentioned above. In this and the remaining tables, all values for measures from Table 4.1 are *average* values per test item, as provided by [incr tsdb()].

4.1.3. Technical Details

I ran all experiments in this chapter on the same dual 500 MHz Pentium III machine with 1 GB of memory. The machine is running *Red Hat Linux 6.2* with the *2.2.15* release of the Linux kernel compiled with *egcs-1.1.2*. Because I had observed that cpu load and availability of main memory have a noticeable effect on cpu time measurements, I made sure the machine was otherwise idle when running the experiments, and sufficient main memory was available so that all data of the process could be in main memory during the experiments, and the operating system did not have to perform unnecessary paging or swapping. PET was compiled using *g++-2.95.2*, set to a high level of optimization (`-O3`); compiler versions and especially optimization levels have a significant effect on performance: Using no optimization in the compiler results in a performance degradation of about 50%. Timing in the parser is done using the `clock(3)` function provided by Unix that reports the approximate processor time used by a given process with a theoretical resolution of 10 ms.

4.2. Setting Out: A Closer Look at Copying

This section sets out to answer empirically a few basic questions about the cost of copying in graph unification that keep reappearing in the literature. Three kinds of unnecessary copying have been identified and defined in previous research (see Section 2.11), but actual amounts have never been quantified. I describe an experiment that determines the amount of early, over, and redundant copying. At the same time, I contrast the practical performance of Wroblewski's and Tomabechi's unification algorithms, and evaluate the advantages of the non-redundant copying schemes discussed in Section 2.9.

Let me summarize the established definitions of the three kinds of unnecessary copying that have been identified.

Over Copying *Copies are made of both dags, and then these copies are ravaged by the unification algorithm to build a result dag. [...] A better algorithm would only*

Unifier	tcpu (s)	tcpu (s)	space (kb)	copying over (nodes)	copying early (nodes)	copying early (nodes)	copying redundant (nodes)
quick check	*on*	*off*	*on*	*on/off*	*on*	*off*	*on/off*
unify1	0.281	1.151	6,672	39.5	226.9	208.2	156.0
unify2	0.205	0.407	5,614	4.2	78.3	52.6	120.6
unify3	0.232	0.406	5,702	9.7	161.0	137.4	126.2
tomabechi	0.170	0.267	5,214	–	–	–	83.6
tom-smart	0.167	0.258	2,174	–	–	–	–

Table 4.3.: Performance and copying behavior of selected unification algorithms on LinGO when parsing the *aged* test set. unify1 and unify2 are the functions of the same name from Wroblewski (1987), unify3 is the function from Section 2.4.2, tomabechi is the algorithm from Tomabechi (1991), and tom-smart adds non-redundant copying from Malouf et al. (2000). The row labelled *quick check* indicates if quick check filtering was enabled or disabled for a column. Where the quick check makes no difference, the column is labelled *on/off*. The static rule filter was enabled in all cases. A plain active chart parser was used.

allocate enough memory for the resulting dag. *(Wroblewski, 1987)*

Early Copying *Copies are created prior to the failure of unification so that copies created since the beginning of the unification up to the point of failure are wasted.* *(Tomabechi, 1991)*

Redundant Copying *[...] a unification result graph consists only of newly created structures. This is unnecessary because there are often input subgraphs that can be used as part of the result graph [...] Copying sharable parts is called redundant copying.* *(Kogure, 1990)*

Early copying is concerned with unnecessary copying in case of a unification failure only, while over and early copying applies in the case of a successful unification.

I used an instrumented version of the cheap parser to quantify the amounts of the three kinds of copying. The amount of early copying is simply the number of nodes that were allocated prior to a failure in unification. The amount of over copying is determined by computing the difference between the number of nodes of the result dag and the number of nodes allocated during unification. Determining the amount of redundant copying is not as straightforward, because there is no tractable way to determine the maximum permissible amount of sharing. A second problem is the overlap in definition between redundant and over copying; the notion of redundant copying subsumes that of over copying. My solution

is to count the total amount of copying for a successful unification, and subtract the amount of copying that takes place in the non-redundant copying scheme of Malouf et al. (2000), using it as the gold standard.

Table 4.3 shows the results of parsing the *aged* test set using the instrumented version of cheap[5]. To show how different the influence of filtering failing unifications on the various algorithms is, I included results with and without quick check (see Section 4.4) where relevant. Let us first look at how much Wroblewski's non-destructive unification method (`unify2`) improves over destructive unification (`unify1`). When not using the quick check, parsing with non-destructive unification is more than twice as fast as destructive unification (64%), when filtering with the quick check is turned on (and the potential benefits from the reduction of early copying in non-destructive unification are smaller), the improvement is still 27%. As discussed by Wroblewski, over copying is not completely eliminated in his approach, but the remaining amount is small, about 4 nodes on average. For comparison, the average total number of nodes in passive edges is 114.6 nodes in this experiment. The amount of early copying is also drastically reduced[6] by about a factor of three. The amount of redundant copying less the amount of over copying should be close to the average number of nodes in a feature structure for `unify2`; this is in fact the case with 116.4 nodes vs. 114.6 nodes. As a further data point, I included a parser that is exclusively using the algorithm from Section 2.4.2 (`unify3`). While the parser using `unify1` has to copy both argument dags before unification, the parser using `unify3` has to copy only one. As could be expected, the resulting performance is about halfway between the parsers using `unify1` and `unify2`.

Tomabechi's algorithm performs only about 17% better than Wroblewski's algorithm when the quick check is enabled, without quick check the advantage grows to 34%. Tomabechi (1991) reports a much greater benefit, but did not use any filtering techniques (while the static rule filter (see Section 3.3) was enabled in my experiments[7]). This resulted in a higher percentage of unifications that failed, where the advantages of Tomabechi's algorithm over Wroblewski's method are most significant. There is an interesting advantage for Tomabechi's algorithm here, namely that restriction of feature structures (see Section 3.3) can be done before copying the result feature structure. This results in significantly less copying, as can be seen by comparing the redundant copying figures for `unify2` and Tomabechi's algorithm without structure sharing improvements (120.6 nodes vs. 83.6 nodes). Comparing the two variants of Tomabechi's algorithm, we can see that the non-redundant copying technique results in only a minor speedup[8] The significant im-

[5]Note that hyper-active parsing was disabled for all unification algorithms, as, in our implementation, it is only available when using Tomabechi's algorithm.

[6]It is not completely eliminated, however, as it would be when using Wroblewski's original definition of early copying.

[7]The static rule filter achieves a filter rate of about 50%.

[8]Malouf et al. (2000) report a more significant speedup in the LKB system. This difference is most likely

provement lies in the reduction of memory usage by more than a factor of two (58%). Looking at the amount of redundant copying shows that on average more than 70% of all nodes can be shared (83.6 nodes out of 114.6 nodes).

We can draw the conclusion that with the high filter rates achieved in current systems using the quick check, unnecessary copying in the case of a unification failure (early copying) is not as much of a problem any more. Unnecessary copying in the case of a successful unification is more significant, and dramatically reduced using Tomabechi's algorithm with structure-sharing improvements. It is also interesting to note that in terms of speed, Wroblewski's algorithm does not perform significantly worse than Tomabechi's algorithm, when effective filtering of failing unifications is applied in the parser.

4.3. Zooming In: How the Parser Spends its Time

We will now look at execution profiles of cheap processing the three grammars obtained with the standard Linux execution-time profiling tool `gprof`. This gives us a detailed picture of how processing time is distributed over the various parts of the parser. Although this distribution roughly agrees for the three grammars, we can draw some interesting conclusions from the places where it differs.

Table 4.4 shows a summary of the `gprof` profiles. The raw `gprof` profiles show the percentage of time spent for each function in the parser, the table groups a number of functions into larger units. Time spent in glb computation is displayed separately, instead of including it in the numbers for unification and quick check vector compatibility checks. The data shows clearly that for all three grammars the majority of time is still spent in unification and copying of feature structures (the first three columns in the table), as has often been observed for unification-based parsers. The actual amount ranges from 55.1% of total run time for the Japanese grammar to 59.3% for the English grammar. Another significant chunk of run time is devoted to the quick check (the next three columns), ranging from 10.5% for German to 18.1% for English.

There is a significant difference in the total amount of run time this adds up to between the three grammars, for two reasons. First, the average time to parse one sentence from the *vm-1*$_{\geq 15}$test set for the Japanese grammar is significantly smaller than that for the other two grammars, see Table 4.6 on page 49 and Table 4.10 on page 58. Thus, time spent in initialization and memory management for parsing a new sentence takes a higher proportion of run time for the Japanese data. Also, feature structures in the Japanese grammar are a lot smaller than in the other grammars, which causes per-unification overhead to play a larger role. Second, the German grammar has a much larger number of rules than the two other grammars; there are 210 rules in the German grammar, compared to 61 and 28

explained by the significantly reduced memory management cost in PET.

Grammar	copy	unify	glb unify	glb qc	qc comp	qc extr	Σ
	%	%	%	%	%	%	%
English	23.3	30.9	5.1	10.1	3.5	4.5	77.4
Japanese	27.1	23.7	4.3	5.9	2.4	6.7	70.1
German	28.1	25.4	4.4	4.7	2.3	3.5	68.4

Table 4.4.: Distribution of run time in cheap when parsing the *fuse*, *vm-1*$_{\geq 15}$, and *balance* test sets, obtained with `gprof`. The column labeled *copy* specifies the amount of time spent in feature structure copying including feature structure memory allocation; the column labeled *unify* corresponds to the amount spent in feature structure unification, excluding glb computation. The column labeled *glb-unify* corresponds to the amount of time spent in glb computation for feature structure unification; the column labeled *glb-qc* corresponds to the amount spent in glb computation for quick check vector compatibility checks. The column labeled *qc-comp* shows the amount of time spent checking quick check vector compatibility, excluding glb computation; the column labeled *qc-extr* shows the amount of time spent in quick check vector extractation. Finally, the last column shows the total percentage of run time recorded by the previous columns. The remaining fraction of run time is spent in the parser proper, including things like initialization and memory management for the parser's data structures.

rules for English and Japanese, respectively. Thus, when parsing with the German grammar, more time is spent in the parser itself, because more tasks are postulated (of which most are filtered by the static rule filter); on average 632,045 tasks are filtered per item for the German grammar, while only 130,691 and 97,061 tasks are filtered for English and German, respectively. This would suggest that better rule indexation in the parser could pay off for the German grammar — currently, this is implemented naively in cheap.

Apart from this, the distribution of run time is comparable in the Japanese and the German grammar; the English grammar, however, deviates in two aspects from the behavior of the other grammars. On the one hand, glb computation for the quick check takes a larger fraction of run time. This could be explained by a smaller effectiveness of the static rule filter for the English grammar, so that more filtering has to be done by the quick check. On the other hand, there is a shift in run time from copying to unification for the English grammar when compared to the others. Looking more closely at the raw profiles permits two interesting observations.

Unification is applied for three different purposes in the parser, (i) to combine a rule and a passive edge, (ii) to combine an active and a passive edge, and (iii) to check if a passive edge spanning the complete input is compatible with any of the start symbols specified by the grammar writer. Table 4.5 shows how the number of calls to unification

Grammar	rule & passive		active & passive		root node	
	% calls	% time	% calls	% time	% calls	% time
English	56.0	34.1	42.2	64.8	1.8	1.1
Japanese	36.0	18.3	55.6	77.4	8.5	4.3
German	36.7	19.0	43.3	70.7	19.9	10.3

Table 4.5.: Three applications for unification in the parser. The column *rule & passive* corresponds to attempts to combine a grammar rule and a passive edge; the column *active & passive* corresponds to attempts to combine an active and a passive edge; the column labeled *root node* corresponds to checking compatibility with one of the start symbols in the grammar, the so called root nodes.

and the time spent in unification are divided among these three categories. We can see that while in the Japanese and German grammars attempts to combine an active and a passive edge dominate attempts to combine a rule and a passive edge, this is much less the case in the English grammar: Looking at the number of calls, the picture is reversed; the distribution of run time is not quite reversed, but a significant shift can be observed. This means that unification fails more often on the key argument of a rule (see Section 3.3) for the English grammar than in the other two grammars. This explains the observed shift from copying to unification time for the English grammar very well, because combining a rule and a passive edge to form an active edge does not necessitate a copy in the hyper-active parsing strategy that cheap is using. While this seems to be a genuine difference in the characteristics of the processing complexity of the grammars[9], the other observation we can make in the table reveals a case where cheap is tuned to specific properties of the LinGO grammar. The cheap approach of using regular unification to check compatibility with the start symbols happens to be cheap enough for LinGO, but the inefficiency inherent in doing full unification without filtering results in a significant performance penalty when processing the Japanese and the German grammars. cheap could easily be improved to apply the standard filtering techniques here.

Execution time profiles are an invaluable tool in optimizing the practical performance of almost any kind of application. The discussion shows that studying profiles of this kind can provide a range of interesting insights into the behavior of a unification-based parser and the properties of the grammars used.

[9]As reported by Flickinger (2000), extra care has been taken in the implementation of the English grammar to reduce the number of active edges that are never extended to a passive egde, e.g. by introducing constraints that are redundant in a linguistic sense.

4.4. Fine Tuning the Quick Check Filtering Method

PET implements the quick check pre-unification filtering technique (Malouf et al., 2000; Kiefer et al., 1999). The quick check uses automatically obtained information about the failure frequency of paths in feature structures to avoid unifications that if executed would fail. The *SING* method proposed by Kogure (1990) exploits the same basic idea: Unification failures occur more often under some features than under others; the quick-check generalizes this idea to paths in feature structures. Uszkoreit (1991) proposes a scheme to add control information to declarative grammars. It allows, among other things, to specify the order in that conjuncts in a feature structure are processed. In this way, unification can be guided so that failures are found early. A proposal by Erbach can be seen as an early variant of the quick check:

> *While in general, there is no way of telling which task is going to fail and which is going to succeed (unless the task is performed!), a large proportion of rule applications can be eliminated by a computationally inexpensive filter. For every rule and every passive item we compute a Prolog term which contains only a subset of the information of its feature structure, i.e., it subsumes the feature structure. Before the task is added to the agenda, the Prolog terms associated with the rule and the items are unified. If this unification fails, we know the task is going to fail, and it is not added to the agenda.*
>
> *(Erbach, 1991b, p. 10)*

The signature-check-based unification filter (Maeda et al., 1994) is another precursor of the quick check. The insight behind all filtering techniques is that the cheapest way to find a failure is not to do (full) unification at all. Under this aspect approaches using context-free filtering (Torisawa et al., 2000) or context-free approximation (Kiefer and Krieger, 2000) can be seen as related.

In PET, for the purposes of quick check vector extraction, the set of quick check paths is represented in an annotated feature structure at run time; this is opposed to a list of paths in other implementations. The representation in a feature structure makes extraction of the quick check vectors computationally cheaper, because many of the paths have common prefixes.

The cheap parser can be configured to collect a set of quick check paths for a given test set. In general, this is done by recording all failure paths when parsing the test set using a modified unification algorithm that continues even after a failure is encountered. Then these paths are sorted by their respective effectiveness, and the best n paths are chosen. The next two sections discuss finding a good measure for effectiveness of a path, and determining the number of paths to use.

4.4.1. Quick Check Path Ordering

The most obvious measure for effectiveness of a failure path is its frequency of occurrence in parsing the test set. This corresponds to assigning a weight of 1 to each occurrence. This measure can be successively improved.

1. When a failure occurs under n paths, assign each of them only a corresponding fraction of the weight, i.e. $1/n$.

2. Do not take into account failures that the quick check could not detect, by checking in the original structures if the information leading to the failure is already there. This is not always the case, as constraints may be unified in during unification, partially expanded paths might be expanded, etc.

3. Make the weight dependent on the cost of finding that failure by full unification. We use the number of nodes visited (recursive calls to the unification function) as a measure for cost of unification. The idea is that some quick check paths only filter unifications that fail very soon, and paths which filter more expensive unifications should be favored. An obvious example is the empty path.

Evaluation of these measures on the *blend* test set demonstrates their effectiveness. The reduction from the base line (using paths computed with the naive measure) to the first measure is 8.5% in parser tasks, and 2.1% in parsing time. The second measure reduces parser tasks by another 7.6%, and parsing time by another 3.8%. The third measure does not improve much upon the previous ones: the reduction in parser tasks is another 0.8%, the reduction in parsing time another 0.5%.

I also experimented with another method to select and rank quick check paths. When processing the test corpus, for each failing unification the set of paths where the failure occurs is recorded. When parsing the test corpus is finished, a minimal list of paths that covers a maximal number of failures can be determined from this information, and ranked so that the number of paths that have to checked to find as many failures as possible is minimized. The quick check paths obtained by this method, however, did not improve upon the paths obtained by using the weight-based method from above.

4.4.2. Determining the Number of Paths to Use

Malouf et al. (2000) discuss the trade-off in choosing the optimal number n of quick check paths. They conclude that n cannot be determined analytically, and report about an experiment to determine n in the LKB for the LinGO grammar. However, for practical reasons, only a subset of the *blend* test set is used, and the variation of n is restricted to a number of support points for the graph.

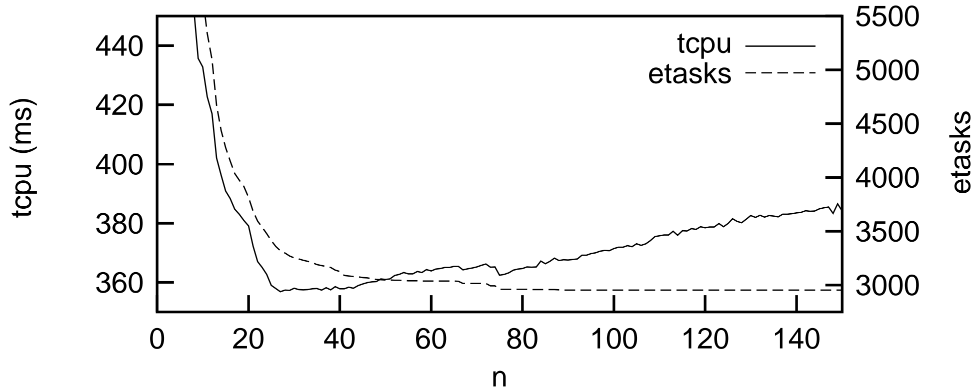

Figure 4.1.: Determining the number of quick check paths. The graph shows *tcpu* and *etasks* average values with the number of quick check paths ranging from 0–150. The results are obtained with cheap on the *blend* test set.

Using PET I could run the experiment on the full *blend* test set trying all n in the range from 0–150 in reasonable time. Figure 4.1 shows the result of this experiment. The results for 0–10 paths, where parsing time quickly drops from 965 ms down to 432 ms are outside the visible part of the graph, since we focus on the minimum of *tcpu*.

The minimum CPU time is at 27 paths, but choosing any number between 25 and 48 paths is no more than 1% worse than the optimum. Even choosing 100 paths results in a performance degraded by only 4%. This means the number of paths can be chosen from a relatively wide range without a significant loss of performance. This was confirmed in experiments on the other test sets. The outcome reflects the findings of Malouf et al. (2000), suggesting the relative speed of type and feature structure unification is comparable between PET and the LKB.

4.5. Partial Expansion and Unfilling

In this section I briefly describe and then evaluate two very effective improvements over making all feature structures well-formed (also called 'expansion') prior to processing (as it is still done in PAGE and LKB). The two techniques are evaluated on LinGO, the Japanese, and the German grammar, using the *fuse*, *vm-1*$_{\geq 15}$ and *balance* test sets.

The first technique, known as *partial expansion*[10], was first explored and found beneficial for the LinGO grammar in the CHIC system (Ciortuz, 2000). Leaf nodes[11] in feature

[10]Closely related lazy evaluation techniques are also discussed in (Götz, 1993; Carpenter and Qu, 1995; Wintner, 1997).

[11]Leaf nodes are nodes without any δ-descendants (Copestake, 2000), or intuitively nodes without sub-

structures are only made well-formed when necessary at run time, that is when a leaf node is unified with a non-leaf node. This technique alone significantly reduces the size of the expanded grammar: as is shown in Table 4.6, the total number of feature structure nodes goes down by 41% for the English grammar, by 58% for the Japanese grammar, and by 43% for the German grammar. In all three grammars the added cost for the delayed unification of constraints at run time is compensated by the reduced size of structures that the system manipulates, resulting in an overall performance improvement between 18% (German) and 25% (Japanese). Looking at the average size of passive edges (*fssize*) shows a reduction of 33% for both LinGO and the Japanese grammar, and of 21% for the German grammar. This demonstrates that the reduction in total grammar size does not necessarily correspond to a similiar reduction at run-time, as not all parts of the grammar are used to the same extent — some parts might not be used at all. There is a significant increase of 31% in the number of attempted unifications (*etasks*) for LinGO, which is due to decreased quick check efficiency: in a partially expanded structure the value for an element in the quick check vector cannot be obtained when the expansion of the structure under the corresponding quick check path has been delayed (for details on the quick check, see Malouf et al. (2000) and Section 4.4). The number of executed tasks increases only slightly (by about 3%) for the Japanese grammar, which could explain the relatively bigger gain in processing speed. Curiously, there is no increase in executed tasks for the German grammar — the quick check paths are not affected by partial expansion for this grammar.

The second technique applied in the PET preprocessor, called *Unfilling* (Götz, 1993; Gerdemann, 1995), goes a step further. After performing (partial) expansion, structures are shrunk again, by recursively removing leaf nodes from the structures. A leaf node under a feature f is removed if its type is the maximal appropriate type of f, and if this node does not introduce structure sharing. It is not removed on root level of the type introducing f. Table 4.6 shows that unfilling reduces the number of nodes (after partial expansion) by 44% for LinGO, by 53% for the Japanese grammar, and by 65% for the German grammar; the average size of passive edges is reduced by 25%, 42%, and 31%, respectively. Again the benefits of smaller structures outweigh the additional cost of expansion at run time significantly for both grammars, resulting in a performance improvement of about 21% for LinGO, 29% for Japanese, and 26% for German.[12]

The overall reduction in the size of the grammar by applying both techniques is 67% for LinGO, and 80% for Japanese and German; the size of passive edges is reduced by 50%, 61%, and 45%, respectively. The corresponding processing speedup is 37% for LinGO, 47% for Japanese and 39% for German. The fact that the technique is least effective on LinGO can partially be explained by an important difference between the

structure.

[12] A nice side-effect is that shrunk structures are also more humanly readable than fully expanded structures, because they are smaller, and the crucial pieces of information become more obvious.

Grammar	Expansion	Grammar size (nodes)	etasks	tcpu (s)	space (kb)	fssize (nodes)
English	*Full*	524,145	5,005	0.958	7,536	329
	Partial	305,836	6,541	0.763	6,257	220
	Unfilling	171,195	6,541	0.603	4,969	166
Japanese	*Full*	786,914	3,994	0.616	5,767	208
	Partial	329,770	4,108	0.459	4,733	139
	Unfilling	153,810	4,108	0.327	3,142	81
German	*Full*	2,374,324	8,740	1.903	13,802	434
	Partial	1,364,861	8,740	1.567	12,660	345
	Unfilling	476,510	8,740	1.158	8,600	238

Table 4.6.: Evaluation of Partial Expansion and Unfilling for LinGO, the Japanese, and the German grammar using cheap with the Tomabechi unifier on the *fuse*, *vm-1*$_{\geq 15}$, and *balance* test sets.

grammars. The LinGO grammar already employs a technique, namely the stipulation of super-types with a minimal set of features (Flickinger, 2000), with effects similar to partial expansion and unfilling, to make processing in LKB and PAGE more efficient. This reduces the potential benefit of partial expansion and unfilling for LinGO. The Japanese and the German grammars have not been manually optimized for processing efficiency in this way.

The practical benefit one can expect from applying the partial expansion and unfilling techniques depends on the particular grammar and can only be determined empirically. Depending on the amount of partial expansion and unfilling that the grammar permits, the increased cost in run-time expansion might even outweigh the benefits of smaller structures for some grammars. My results on three significantly different grammars suggest, however, that these techniques will be beneficial for typical large-scale unification grammars.

4.6. Feature Structure Encoding Techniques

This section presents a detailed empirical comparison between different ways of feature structure encoding for variants of the Tomabechi graph unification algorithm, contrasting an encoding based on unordered lists of feature-value pairs (representative for many LISP-based implementations of graph unification) with two encodings that represent features by position (reminiscent of WAM-like fixed-arity representations). All variants have been

implemented in PET, and were evaluated using the English and the Japanese grammar[13].

This particular issue of representation has been an area of active research in the Prolog and abstract machine community (see Section 2.12), and encodings based on fixed arity are assumed throughout, as in the WAM (Warren, 1983a; Aït-Kaci, 1991), $\mathcal{AMALIA}$ (Wintner, 1997), and LiLFeS (Makino et al., 1998). An interesting exception is the abstract machine suggested for order-sorted feature term unification (Aït-Kaci and Di Cosmo, 1993), where feature-value lists are used; this encoding has been adopted in CHIC (Ciortuz, 2000). In the research on graph unification, however, there has been no detailed investigation of feature structure representation issues, instead a straightforward feature-value list encoding is often assumed (Wroblewski, 1987; Kogure, 1990; Emele, 1991; Tomabechi, 1991; Malouf et al., 2000).

4.6.1. Encodings Based on Fixed Arity

The feature structure representation suggested in (Tomabechi, 1991) uses an unordered list of feature-value pairs to represent the features and values of a given node. This requires a linear search in this list every time the value of a given feature is accessed. In our formal framework however, each type t has an associated fixed set $Appfeat(t)$ of appropriate features (*fixed arity*). Moreover, the set of appropriate features is typically small. It seems reasonable to use an encoding where features are not represented explicitly, but instead use an array containing only the values (I will call this a *value array*), and represent features implicitly by position in the array. The value array has a fixed size and layout determined by the type. This encoding scheme makes searching for features unnecessary, thus reducing the access cost for a feature to constant time. In addition the space required to represent a node is reduced, since features are no longer explicitly represented.

We now need to find a layout of features for each type, so that all features appropriate to this type can be represented.[14] We can think of a layout as a partial function $\pi : (Type, Feat) \rightarrow Pos$ that assigns to all the features appropriate for a type a position in its value array. A valid layout must assign a distinct position to each feature of a type, i.e. $f_1 \neq f_2$ iff $\pi(t, f_1) \neq \pi(t, f_2)$ for all $f_1, f_2 \in Appfeat(t)$. The maximal position assigned to a type determines the size required for a value array for a node of this type, the size of the array is bounded from below by the type's number of appropriate features.

There are two conflicting desiderata when determining the feature layout. On the one hand we want to minimize the space consumption for each type, on the other hand we want to minimize the number of translations between different layouts, so-called *coercions*. A

[13] Unfortunately, the German grammar was not yet available in the common formalism when this experiment was carried out.

[14] Closely related problems are discussed in research on method dispatch techniques in object oriented programming languages with multiple inheritance; for a recent survey see (Driesen, 1999).

coercion is necessary when during unification of two nodes the input types and the result type do not have the same encoding layout.

One extreme, where coercions are eliminated altogether, is to guarantee for each pair of compatible types t_1 and t_2, with $t_3 = t_1 \sqcap t_2$, that $\pi(t_1, f) = \pi(t_2, f) = \pi(t_3, f)$ for all $f \in \mathit{Appfeat}(t_3)$. A trivial (but extremely wasteful) way to ensure this is to globally assign a unique position to each feature, such that $\pi(t_1, f) = \pi(t_2, f)$ for all types t_1, t_2. A more manageable solution that also eliminates coercions is to partition the types into sets of types that do not interfere in unification, and then assign unique positions to all features appropriate to the types in each set independently. This approach is described in Section 4.6.3.

The other extreme is to assign an independent layout for each type, by locally assigning a position for each feature appropriate to a type. In this setup the space required for representing a type is minimized, but a coercion is required in all unifications involving two different types. A straightforward improvement of this approach that also guarantees a minimal encoding is to identify sets of types sharing the same set of appropriate features (a *feature configuration*), and then assign the same layout to all types with the same feature configuration. The number of feature configurations is typically much smaller than the number of types, because many types do not introduce new features. This approach is discussed in Section 4.6.4.

4.6.2. Adapting Tomabechi's Unification Algorithm

I had to adapt Tomabechi's unification algorithm for my fixed-arity-based encodings. In Tomabechi's original algorithm, a list-valued *comp-arc-list* slot is used to record feature-value pairs temporarily added during unification. If we want to both take advantage of the fixed arity encoding (constant access time to features), and at the same time avoid allocation during unification we would need a preallocated second value array to hold the temporary values. This would almost double the size of nodes for most types, since the value array dominates the space to represent a node. Thus, I decided to omit the *comp-arc-list* slot, and instead reintroduce some early copying. As discussed in Section 4.2, early copying constitutes only a minor problem in our setup, since most failing unifications are filtered by cheap filters like the quick check (see Section 4.4). Whenever we would have to use the *comp-arc-list* slot in Tomabechi's original setup, instead we allocate a new node and (temporarily) forward both input nodes to the new node. Experiments at this point showed that additionally eliminating the *new-sort* slot in the same way would not significantly increase early copying (this is because in our setup, in most cases where the *new-sort* slot is used, the *comp-arc-list* is also used), so I apply the same technique here (allocate a new node, and forward the input nodes). The only remaining temporary slot

(sometimes called 'scratch' slot), protected by a generation mark, is the *forward* slot.[15]

I implemented three variants of this algorithm, using three different encodings: one using the partitioned fixed arity encoding, a second one using the minimal fixed arity encoding, and a third using the plain feature-value list encoding. The next two sections describe the two fixed-arity-based encodings.

4.6.3. The Partitioned Fixed Arity Encoding

The first fixed arity encoding I implemented avoids coercions by partitioning the type hierarchy into independent blocks, and using the same layout of features for types in a common block. The partitioning is set up such that unification of two types with a non-empty set of appropriate features in a common partition either fails, or results in a type from the same partition. This property makes the implementation of the unification algorithm simple and efficient: whenever two nodes with compatible types are unified, we know they have the same layout, and so has the resulting node. Thus no search and no translation of positions is necessary for the recursive unifications of the shared arcs of two given nodes to be unified. The drawback of this encoding is the potentially large amount of space wasted: each type's value array has to have room for all the features of types that it could potentially be refined to.

The partitioning is computed in the preprocessor, using a simple algorithm that works bottom-up to split the hierarchy into suitable partitions, starting at leaves in the hierarchy. The algorithm starts by putting each type into a partition of its own, and then successively merging partitions. We visit each type that is a leaf in the hierarchy, and recursively merge all parents of this type into the same partition, stopping the recursion at types that have an empy set of appropriate features.

This simple scheme works reasonably well for LinGO and the Japanese grammar. The LinGO grammar has a total of 155 features. It is split into 23 partitions, the maximal number of features in a partition is 56 in one partition. The Japanese grammar has a total of 72 features. It is split into 19 partitions, the maximal number of features in one partition is 22.

4.6.4. The Minimal Fixed Arity Encoding

My second fixed-arity-based encoding minimizes the space required to represent each type. Each type's value array has room for exactly its appropriate features. To avoid useless coercions at run time, we first compute all distinct feature configurations. There are 131 such configurations for the LinGO grammar and 27 for the Japanese grammar.

[15] The resulting algorithm is similar to Wroblewski's `unify2` (Wroblewski, 1987), with the important difference that I use the structure sharing improvements from (Malouf et al., 2000).

Each configuration is assigned a unique identifier. The mapping from types to feature configuration identifiers is held in an array at run time, so we can efficiently check if two types have the same feature configuration. When two types t_1 and t_2 with different feature configurations are unified, we need to translate feature positions. This is done using a table $\Pi[\mathit{Featconf}, \mathit{Feat}] \rightarrow \mathit{Pos}$ that contains the position corresponding to a given feature configuration and feature, and a second table $\Phi[\mathit{Featconf}, \mathit{Pos}] \rightarrow \mathit{Feat}$ containing the feature given a feature configuration and a position. When t_1 and t_2 with feature configurations c_1 and c_2 are unified, we iterate over all positions valid for t_1. For each position i in t_1 we compute the corresponding position in t_2 by the table lookup $\Pi[c_2, \Phi[c_1, i]]$.

4.6.5. Comparison and Discussion

I evaluated the different variants on LinGO and the Japanese grammar. A total of four configurations were compared, the original Tomabechi algorithm using the feature-value list encoding, the adapted algorithm using the feature-value list encoding, the adapted algorithm using the partitioned fixed arity encoding, and the adapted algorithm using the minimal fixed arity encoding.

Table 4.7 shows the results of processing the *fuse* and *vm-1*$_{\geq 15}$test sets. The differences in processing time are rather small, for LinGO the fastest unifier is only 5.9% faster than the slowest one, for the Japanese grammar the maximal speedup is 4.9%. This result is interesting in itself, since a much higher variation could be expected given the considerable differences in algorithms and encoding. On the other hand, the difference in space consumption is much more significant, the maximal reduction is 48% for LinGO, and 45% for the Japanese grammar.

Looking at the cpu time rankings reveals an interesting difference in behavior on the two grammars. The partitioned fixed arity encoding performs best on the Japanese grammar, while it is the worst performer on LinGO. Comparing the memory consumption rankings shows they are identical for both grammars, offering no explanation for the difference in relative speed. We can look at the amount of space wasted (see Section 4.6.3) by the encoding at run time[16]; it is 8,544 kb for LinGO and only 10 kb for the Japanese grammar. This difference would very well explain the bad performance of the partitioned encoding on LinGO. By taking a closer look at how the LinGO hierarchy is partitioned by my scheme, I found there is one partition containing a very large number of features (56), while all the other partitions are comparatively small. This partition contains the subhierarchy of semantic relations – a large number of different features is introduced in this part of the hierarchy, and my simple partitioning scheme merges them all into one par-

[16] This was obtained using an instrumented version of the unifier that counts the number of bytes wasted for encoding purposes on each allocation of a feature structure node during unification.

Algorithm	Encoding	English				Japanese			
		tcpu		*space*		*tcpu*		*space*	
		(ms)	#	(kb)	#	(ms)	#	(kb)	#
original	*list*	584	3	4,683	2	328	3	2,986	2
adapted	*list*	569	2	6,639	4	329	4	5,267	4
adapted	*fix-part*	590	4	5,067	3	313	1	3,005	3
adapted	*fix-min*	555	1	3,436	1	323	2	2,852	1

Table 4.7.: Evaluation of four different unifiers for LinGO and the Japanese grammar using cheap on the *fuse* and *vm-1*$_{\geq 15}$ test sets. Sub-columns labelled '#' indicate relative ranks within the respective column.

tition. To verify my explanation I tried the four unifiers on a version of LinGO without semantics (by removing all information under the path SYNSEM.LOCAL.CONT), and indeed found my hypothesis supported: for this semantics-free version of the grammar the partitioned fixed arity encoding performs best, closely followed by the minimal fixed arity encoding.

The overall best performer on the two grammars considering both time and space requirements is the minimal fixed arity encoding. It consumes significantly less memory than the three other configurations, and is always among the two fastest configurations in my experiments. The partitioned fixed arity encoding relies on a good partitioning, which cannot always be found using my simple scheme. Improvements to the partitioning scheme could easily be devised, but cannot be expected to result in an interesting overall performance improvement. My experiments show clearly that chosing a fixed-arity-based feature structure encoding over a feature-value-list-based encoding does not necessarily make an interesting difference in processing speed.

4.7. Copying, Recomputation and Trailing in Active Chart Parsing

This section takes a look at three different ways for an active chart parser[17] to deal with the feature structures associated to active edges. The standard approach is to copy the associated feature structure when an active edge is derived, so that it is available when attempts are made to extend this active edge. An improved strategy, called *hyper-active* parsing, is proposed in Oepen and Carroll (2000b). Based on the observation that unifica-

[17] I consider *hyper-active* parsing (Oepen and Carroll, 2000b) an instance of active chart parsing for the purpose of the current discussion.

Variant	**tcpu** (ms)	**space** (kb)
active	294.29	3,951
hyper-active	285.43	2,581
hyper-active with trailing	273.24	2,547
hyper-active with unused trailing	301.30	2,597

Table 4.8.: Comparison of active parsing and hyperactive parsing with different rebuild strategies on LinGO, using the *fuse* test set.

tions are less expensive than copies, the hyper-active parsing strategy trades unifications for copies: when an active edge is derived, the associated structure is not copied, instead it is rebuilt whenever it is needed again. The benefits of this strategy depend on how often the structure associated to an active edge needs to be rebuilt; given the highly effective pre-unification filters this number is low (around two for the LinGO grammar), and thus the hyper-active strategy is effective.

An alternative way of implementing hyper-active parsing is using a scheme similiar to the skeleton/environment representation discussed in Section 2.2. The idea is that instead of redoing the unification necessary to obtain the feature structure of an active edge, an *environment* is created when the active edge is first derived. This environment represents all the changes that need to be applied to the input structures to obtain the feature structure of the active edge. The environment is created using a technique called *trailing*. When performing unification, all changes to the input feature structure are saved in a data structure called the *trail*. After unification, the environment is constructed from the information in the trail.

The three variants (active, hyper-active and hyper-active with trailing) are implemented in PET for the Tomabechi unifier. I compare their performance on the LinGO grammar, using the *fuse* test set. To allow estimating the overhead of environment construction, I included[18] a fourth variant that constructs the environments, but does not make use of them, i.e. it uses regular unification to rebuild the feature structures of active edges. The results of the experiment are shown in Table 4.8. I verified that all four variants indeed compute the same results. The reduction in parsing time from active to hyper-active parsing is 3.0%, using environments results in a further improvement of 4.3%. The penalty for constructing the environments, but not using them, is 5.6%. Looking at space consumption, we see a significant reduction of almost 35% from active to hyper-active parsing. The space consumption of the three hyper-active variants is (as to be expected) almost identical.

While the differences in parsing time visible from the outside seem small, a look in-

[18]Following a suggestion by Stephan Oepen.

Variant	Total run time	unify	copy	env::build	env::apply
active	741.2 s	145.5 s	331.9 s	–	–
hyper-active	722.7 s	215.1 s	223.3 s	–	–
trailing	691.4 s	164.6 s	215.8 s	16.7 s	15.0 s

Table 4.9.: Distribution of run time in the unifier for the active, hyper-active and trailing hyper-active parsers. Absolute times are shown.

side shows more dramatic differences. Table 4.9 shows information about the distribution of run time inside the unifier obtained from `gprof` profiles (see Section 4.3). I show absolute times to enable direct comparison. Looking at the time spent in unification and copying shows that the hyper-active parser successfully trades unifications for copies: the hyper-active parser spends about 70 seconds longer in unification than the active parser, but saves 109 seconds in copying. Some of these savings are, however, cancelled out by higher overhead in other parts (most noticeable in quick-check vector extraction, which is more expensive in the hyper-active parser, as the quick check vector has to be extracted from the intermediate unification result rather than from the copy), resulting in an overall improvement of 18 seconds. Looking at the trailing parser, we can see that it spends 19 seconds longer in unification than the the active parser, while executing the same unifications. This is caused by the overhead of trailing. Another 17 seconds are spent by the trailing parser in environment construction; application of environments takes another 15 seconds. Still, this results in a sum of only 196 seconds, compared to the 215 seconds the hyper-active parser spends in unification. The trailing parser also uses a slightly different version of copying[19] that saves 8 seconds compared to the active parser. The overall improvement of the trailing version over the plain hyper-active parser is 31 seconds (4.3%). The profile shows that a total of 1,087,592 environments are constructed, and a total of 2,536,573 environment applications take place. This means an environment is used only about 2.3 times on average. This explains why the cost of environment construction does not pay off very well, resulting in the comparatively minor overall speedup.

My experiment supports the results reported in Schulte (1999). Schulte compares trailing and copying for constraint programming. He discusses three approaches to making previous computation states available in search: copying, recomputation and trailing. This directly relates to the three parsers that I compared: the active parser uses copying to make active edges available, the hyper-active parser uses recomputation, and the trailing hyper-active parser uses trailing; however, I do not use trailing to undo changes, but to redo changes. Schulte concludes that a system based on copying is competetive compared to

[19] The copying function in the trailing parser cannot reuse top-level arcs, resulting in slightly higher memory usage, but lower execution time.

trailing-based systems. Using a combination of copying and recomputation can even outperform trailing-based systems. My data shows that the improvement from trailing is only minor. I can support Schulte's observation that design and implementation complexity is much higher for a trailing system: every update operation needs to take trailing into account. While the complexity was reduced by encapsulating update operations, the trailing implementation proved to be hard to debug.

Although these experiments would predict that the trailing-based approach could pay back in a setup where environments were used more frequently, such a setup would conflict with the underlying assumption of the hyper-active parsing strategy. It can be expected that within the window where hyper-active parsing pays off, no significant performance improvement can be achieved by a trailing-based approach. Given the small overall practical performance improvement, and the high cost of implementing and maintaining the trailing variant, I decided to exclude it from further development. The hyper-active parsing strategy with recomputation by regular unification is now the standard in PET, and has proven to be beneficial on the LinGO, Japanese and German grammars.

4.8. Cross-comparison: Comparing Across Grammars and Platforms

In this section I review the performance of PET on test sets for the three grammars and compare results with LKB data obtained on the same grammars and test sets. I try to identify a measure to assess both parser performance abstracting from grammar complexity, and grammar complexity abstracting from parser performance. This would allow predicting practical performance for a given platform and grammar from existing data obtained on a different platform and grammar. Such a prediction could then be used, for instance, to verify if a processing system behaves as expected on a given new grammar, and to identify unexpected performance leaks[20].

Table 4.10 shows the raw data underlying the discussion. It summarizes relevant information from the performance profiles for the LKB and PET on the three grammars. As expected, the number of passive edges (*pedges*) agrees between the LKB and PET in all cases. The filter rates, and as a result the number of executed tasks (*etasks*), also agree roughly[21] The same goes for the number of successful tasks (*stasks*) and the number of unifications (*unifs*); the small differences can be explained by a slightly different way to count successful tasks in lexical processing. Significant differences between the two processing systems are in the average parse time (*tcpu*) and the amount of allocated space

[20]The idea underlying the discussion in this section was developed in cooperation with Stephan Oepen; an extended discussion of this topic is provided in Oepen (2001).

[21]There is no complete agreement, because quick check efficiency is slightly reduced in PET due to partial expansion (see Section 4.5).

Grammar	System	tcpu (s)	filter (%)	etasks	stasks	unifs	pedges	dspace (kb)
English	*LKB*	3.28	96.1	5,946	2,695	8,840	1,850	16,894
	PET	0.59	95.2	6,541	2,661	8,890	1,850	5,290
Japanese	*LKB*	0.60	95.5	950	851	1,300	725	4,053
	PET	0.07	95.8	893	851	1,190	725	752
German	*LKB*	6.88	98.7	8,781	5,387	11,910	3,238	37,099
	PET	1.10	98.7	8,438	5,001	12,064	3,238	9,031

Table 4.10.: Performance profiles of the LKB and PET on LinGO, the Japanese, and the German grammars for the *fuse*, *vm-1* and *balance* test sets, restricted to compatible items.

(*dspace*). PET is faster than the LKB, between a factor of 5.6 for LinGO and a factor of 8.6 for the German grammar. PET also allocates less memory, between a factor of 3.2 for LinGO and a factor of 5.4 for the Japanese grammar. What we can conclude from this table is that PET is a factor of 6.8 faster than the LKB, on average. This allows us to predict performance of one platform on a given grammar, provided we have information on the performance of the other platform on the same grammar.

But what can we do if that information is not available? Imagine, for the moment, we only have information about the performance of PET on the LinGO grammar, and the LKB on the Japanese grammar. When comparing absolute values only, with an average parse time per test item of 0.59 s PET does not seem to be faster than the LKB with an average time of 0.60 s. This, of course, is due to the different processing complexity of the two grammars and test sets. The number of executed tasks gives us a handle on the processing complexity. We can see that about 6 times as many tasks are executed per test item for the English input when compared to the Japanese data. Under the simplified assumption that the number of tasks is in direct proportion to the processing complexity of a grammar and test set, the number of tasks a given processing system executes per second should be independent of the grammar and test set. This is roughly the case for our example: PET executes around 11,086 tasks per second on LinGO and 13,132 tasks per second on the Japanese grammar; the LKB executes 1,811 and 1,583 tasks per second, respectively. In other words, PET can execute about 7 times the number of tasks that the LKB can execute in the same time. In our example we would now predict the average parse time of PET on the Japanese input as 0.09 s (obtained by dividing the LKB's processing time of 0.60 s by 7), which is close to the actual value of 0.07 s. Remember that we made this prediction without ever referring to a direct comparison between the two processing systems on the same grammar.

	English	Japanese	German	
LKB	1,811/s 7.3 13,132/s	1,583/s 4.8 7,650/s	1,277/s 8.7 11,086/s	**PET**
PET	11,086/s 7.0 1,583/s	13,132/s 10.3 1,277/s	7,650/s 4.2 1,811/s	**LKB**
	Japanese	**German**	**English**	

Table 4.11.: Cross-comparison of PET and LKB performance for the three grammars. Performance is expressed in terms of the number of tasks executed per second of cpu time (*etasks / s*). Each cell in the table contains three pieces of information: In the upper left corner, the performance of the combination of grammar and processing system that is identified by referring to the top row and the leftmost column is shown; the lower right corner of each cell shows the performance for the combination that is identified by referring to the bottom row and the rightmost column. Centered in each cell is the resulting performance ratio for the two combinations.

Table 4.11 systematically shows the performance ratios that we obtain in this way. If *etasks / s* was in direct proportion to grammar and test set complexity, all ratios should be identical; this is not the case, and the reason is quite obvious: The cost of executing one task varies between grammars, for a number of reasons like differences in the average size of a feature structure. We can conclude from the data in the table that the cost of executing one task is comparable between the English and Japanese grammars; for the German grammar that cost is significantly higher. Curiously, PET is affected from this higher cost to a larger degree than the LKB — this seems to be another case where PET exhibits a certain amount of tuning to the specifics of the LinGO grammar, similar to what we found in Section 4.3.

5. Conclusions and Outlook

5.1. Summary

In this thesis, I have explored a number of central practical performance properties of unification-based parsing on large-scale grammars, building on precise empirical data obtained with the PET platform. Through empirical study, I was able to answer a number of open questions in the area of unification-based parsing; most importantly I was able to show that the question of feature structure encoding (list-based vs. fixed-arity-based) is less relevant than is often assumed. Other important results include the quantification of the different types of unnecessary copying on a large-scale grammar, an analysis of the distribution of run time in the parser, the evaluation of two very effective techniques for reducing the size of feature structures (partial expansion and unfilling), a study of three different strategies to handle feature structures associated with active edges, and finally a discussion of how to predict practical performance across platforms and grammars. In line with the results of Erbach (1991c); Carroll (1994); Oepen and Carroll (2000a), I claim that empirical study is essential in implementing efficient unification-based parsers; also, empirical study is the most effective way to find the right focus for work on the improvement of algorithms and the development of new techniques.

The second major contribution of this thesis is the implementation of the PET platform itself. PET synthesizes a significant body of existing experience in efficient unification-based processing, in the form of a carefully implemented modular set of efficient building blocks that allow both easy implementation of efficient processors, and systematic, controlled experimentation with unification-based processing strategies in a common context. The cheap parser of the PET platform achieves a very attractive practical performance. Both time and space requirements are significantly reduced compared to the PAGE and LKB systems; the process size is reduced by an order of magnitude. PET has been quite successful: it is now the standard run-time system for test set processing used by the developers of the three large-scale HPSG grammars in the consortium, and it is also employed in product development in at least two different companies, marking the first time large-scale HPSG grammars are being used in commercial product development[1].

[1]See Flickinger et al. (1985); Proudian and Pollard (1985) for discussion of the earliest work on grammar

Another contribution is the discussion of an efficient implementation of a *semi-lattice* construction algorithm. Automatic *semi-lattice* construction is an important device for allowing straightforward specification of the grammar, while still enabling efficient processing. In addition, within a comprehensive review of previous work on efficient graph unification, I identified and solved a previously overlooked problem with one of the foundational algorithms for graph unification.

5.2. Quantifying Progress

I take a wider perspective now and provide an idea of the overall progress made in processing the LinGO grammar over a period of four years. The oldest available [incr tsdb()] profiles (for the *aged* test set) were obtained with PAGE (version 2.0 released in May 1997) using the October 1996 version of LinGO. I contrast this with today's best parsing performance on the current reference version of LinGO (May 2000). All data were sampled on the same 300 MHz UltraSparc server.

Grammar	Platform	readings	filter (%)	etasks	pedges	tcpu (s)	space (kb)
October 1996	*PAGE*	2.55	51.3	1,763	97	36.69	79,093
January 2001	*PET*	13.53	94.2	1,571	439	0.24	1,142

Table 5.1.: Progress made in processing the LinGO grammar over four years. Numbers obtained on a 300 MHz UltraSparc using the *aged* test set.

Table 5.1 shows that average parsing times[2] per test item have dropped by more than two orders of magnitude (a factor of 150 on the *aged* data), while memory consumption was reduced by a factor of more than 60. Because in the early PAGE data the quick check filter was not available, current filter rates are much better and result in a reduction of executed parser tasks. At the same time, comparing the number of passive edges licensed by the two versions of the grammar provides a good estimate on the search space explored by the two parsers. The *aged* data shows an increase by a factor of 4.5. Assuming that the average number of passive edges is a direct measure for input complexity[3] (with respect to a particular grammar), I extrapolate the overall speed-up in processing the LinGO grammar as a factor of roughly 700.

implementation in the HPSG framework in an industrial research setting.

[2]The *tcpu* values for PAGE include garbage collection time, which is eliminated in PET.

[3]This assumption is supported by very strong linear correlation between the number of passive edges and parsing time in both profiles ($r^2 = 0.92$ for the PAGE data; $r^2 = 0.99$ for the PET data).

5.3. Open Questions and Future Work

From the perspective of efficient processing, the most pressing open question emerging from this thesis is a comparison of the approach to unification-based parsing discussed here with approaches that are based on compilation. Given the high unifier throughput reported in Miyao et al. (2000) for LiLFeS, it would be interesting to see if the techniques for efficient processing discussed in the present work can be integrated into LiLFeS. An evaluation of the techniques from Penn (2000) on a large-scale grammar would also be highly interesting, given the promising results reported by Penn for a smaller grammar.

A number of open questions regarding the application of HPSG-based grammars can be tackled fruitfully only now that the efficiency problem in processing has been solved for many research and commercial applications through the joint efforts of the consortium. Central issues within such applications include both robustness and resolution of ambiguity.

A. The Input Language

The PET preprocessor is designed to accept a conjunctive subset of $\mathcal{TDL}$ as input, excluding the various forms of disjunction and negation. The recognized subset of $\mathcal{TDL}$ is comparable to that of the LKB system, adding the ability to handle $\mathcal{TDL}$ templates.

Given below is the specification of the input syntax in BNF form. This is adapted from the definitions given in the $\mathcal{TDL}$ reference manual (Krieger and Schäfer, 1994b), with changes inspired by the LKB documentation (Copestake, 1999). The main difference to the $\mathcal{TDL}$ reference manual is the ommision of disjunctions and negations, and the correction of a number of obvious, mostly minor, bugs in the BNF.

$\mathcal{TDL}$ Main Constructors

start → [*block-or-statement-list*]
block-or-statement-list → *block-or-statement* [*block-or-statement-list*]
block-or-statement → *block* | *statement*
block → *domain-block* | *instance-block* | *type-block*
domain-block → '`begin :domain`' *domain* '`.`' *start* '`end :domain`' *domain* '`.`'
domain → *atom*
instance-block → '`begin :instance.`' *start-or-instance-list* '`end :instance.`'
start-or-instance-list → *start-or-instance* [*start-or-instance-list*]
start-or-instance → *start* | *instance-def*
template-block → '`begin :template.`' *start-or-template-list* '`end :template.`'
start-or-template-list → *start-or-template* [*start-or-template-list*]
start-or-template → *start* | *templ-def*
type-block → '`begin :type.`' *start-or-type-list* '`end :type.`'
start-or-type-list → *start-or-type* [*start-or-type-list*]
start-or-type → *start* | *type-def*

Type and Instance Definitions

type-def → *type avm-def* '`.`' | *type subtype-def* '`.`'
type → IDENTIFIER

subtype-def → ':<' *type* [*status*]
avm-def → ':=' *conjunction* [*status*]
status → ',status:' IDENTIFIER
conjunction → *term* | *term* '&' *conjunction*
term → *type* | *atom* | *feature-term* | *coreference*
atom → STRING | INTEGER | '''IDENTIFIER
feature-term → '[' [*attr-val-list*] ']'
attr-val-list → *attr-val* [',' *attr-val-list*]
attr-val → *attr-list conjunction*
attr-list → *attribute* ['.' *attr-list*]
attribute → IDENTIFIER | *templ-par*
coreference → '#' IDENTIFIER
instance-def → *instance avm-def*
instance → IDENTIFIER

Lists and Difference Lists

There is some syntactic sugar to ease the notation of lists and difference lists.

term → *list* | *diff-list*
diff-list → '<!' [*conjunction-list*] '!>'
conjunction-list → *conjunction* [',' *conjunction-list*]
list → '<' *conjunction-list* '>' |
'<' *conjunction-list* ', ...' '>' |
'<' *conjunction-list* '.' *conjunction* '>'

Parametrized Templates

TDL allows the definition and instantiation of parametrized templates. The *TDL* reference BNF also allows template parametes to be used for attributes. This is also supported in PET.

term → *templ-par* | *templ-call*
templ-call → '@'*templ-name* '(' [*templ-par-list*] ')'
templ-name → IDENTIFIER
templ-par-list → *templ-par* [',' *templ-par-list*]
templ-par → '$'*templ-var* ['=' *conjunction*]
templ-var → IDENTIFIER
templ-def → *templ-name* '('[*templ-par-list*]')'':=' *conjunction* '.'

$\mathcal{TDL}$ Statements

Except for the `include`-statement, all the statements are ignored by the preprocessor. They are only recognized so the input files can be read without changing them.

statement → '`defdomain`' *domain* '`.`' | '`deldomain`' *domain* '`.`' |
'`expand-all-instances.`' |
'`include`' *filename* '`.`' |
'`leval`' COMMON-LISP-EXPRESSION '`.`'
filename → STRING

B. Binary Representation of Grammars

This appendix specifies the intermediate form of a grammar that is procuced by the flop preprocessor from the $\mathcal{TDL}$ grammar source, and read by the processor at run time. This intermediate form is a compact binary file containing all the relevant information from the grammar sources (i.e. the type hierarchy, type constraints, the lexcion and grammar rules) in a form geared to easy usability by run time processors. It is an ideal starting point for systems that want to process grammars in our common descriptive formalism (Oepen et al., 2000), avoiding idiosyncrasies of the grammar source format, and without implementing the otherwise necessary preprocessing steps. This can save significant development time, as exemplified by the cali system (van Lohuizen, 2001), that was was adapted to support this formalism in just a few weeks, starting from the flop preprocessor output.

I give a semi-formal specification of the file format. The file consists of sections, I recursively decompose each section into smaller elements, down to atomic elements of the file format. Atomic elements are *int*, *short* and *char*, composed of four bytes, two bytes and one byte, respectively. *int* and *short* elements are stored in little endian byte order.

grammar-file

A grammar file consists of eight sections in a fixed order. The *header* contains information identifying the file format and the name of the grammar contained in the file. The *toc* contains the offsets to later sections in the file, so that it is possible for a program to skip over sections it is not interested in. The *symbol-tables* contain the names for the objects of the grammar. The *hierarchy* gives all information about the type hierarchy underlying the grammar. The *feature-tables* contain information about the encoding scheme for types, and appropriateness information for attributes. The *rules* section contains all the grammar rules, the *lexicon* section contains the (full-form) lexicon. Finally, the *constraints* section contains the feature-structure constraints of types and instances.

type	description
header	Header to identify file
toc	Offsets to sections in file
symbol-tables	Symbols in the grammar
hierarchy	Type hierarchy
feature-tables	Additional feature information
rules	Grammar rules
lexicon	Full-form lexicon
constraints	Constraints of types and instances

header

type	identifier	description
int	`magic`	Magic value to identify file format
int	`version`	Version of file format
string	`description`	Description (name) of the grammar

string

type	identifier
short	`length`
`length` × *char*	`text`

toc

type	identifier
int	`offset-hierarchy`
int	`offset-feature-tables`
int	`offset-rules`
int	`offset-lexicon`
int	`offset-constraints`

symbol-tables

`ntypes` is `npropertypes` + `nleaftypes` + `nsymbols` + `ninstances`.

type	identifier
int	`npropertypes`
int	`nleaftypes`
int	`nsymbols`
int	`ninstances`
int	`nattrs`
`ntypes` × *string*	`type-name`
`nattrs` × *string*	`attr-name`

hierarchy

type	identifier
int	`nbits`
`npropertypes` × *bitcode*	`type-bitcode`
`nleaftypes` × *int*	`leaftype-parent`
`ninstances` × *int*	`instance-parent`

bitcode

type	description
bitcodepart	Parts of a bitcode
...	Repeated until end marker
`int` = 0	End marker
`short` = 0	End marker

bitcodepart

type	identifier
int	`value`
`value` $\neq 0$: *short* $\neq 0$	`repetition`

feature-tables

`n` is `npropertypes` + `nleaftypes`.

type	identifier
int	`encoding-type`
`n` × *type-feat-info*	`type-feat-info`
int	`nfeatsets`
`nfeatsets` × *featset*	`featset`
`nattrs` × *short*	`appropriate-sort`

type-feat-info

type	identifier
int	`featset`
char	`waste`

featset

type	identifier
short	`nattr`
`nattr` × *short*	`attr`

rules

type	identifier
int	`nrules`
`nrules` × *rule*	`rule`

rule

type	identifier
short	`type`
char	`arity`
char	`key-daughter`
char	`head-daughter`

lexicon

type	identifier
int	`nles`
`nles` × *lexicon-entry*	`lexicon-entry`

lexicon-entry

type	identifier
short	`preterminal-type`
short	`affix-type`
char	`infl-pos`
char	`nstems`
`nstems` × *string*	`stem`

constraints

`n` is `npropertypes` + `nleaftypes`.

type	description
`n` × *constraint*	Constraints associated to types
`ninstances` × *constraint*	Constraints associated to instances

constraint

type	identifier
int	`nnodes`
int	`narcs`
`nnodes` × *node*	`node`

node

type	identifier
short	`type`
short	`nattrs`
`nattrs` × *arc*	`arc`

arc

type	identifier
short	`attribute`
short	`value`

C. PET Usage

The Preprocessor

The basic usage of the preprocessor is simple. A file containing the settings specific to the grammar to be converted has to be created. Then the preprocessor is invoked from the directory containing the grammar sources, giving the main grammar file (`english.tdl` for LinGO) as argument. The output goes to a corresponding `.gram` file (replacing the `.tdl` extension by `.gram`). As an example, preprocessing the LinGO grammar is done by this command:

```
(~/lingo) $ flop english.tdl
```

This will result in a file called `english.gram` in the same directory. The output for the LinGO grammar looks like this:

```
reading 'Version.lisp'...
converting 'english.tdl' (LinGO (eubp)) into 'english.gram' ...
loading 'english.tdl'... including 'fundamentals.tdl'... 'lextypes.tdl'... 'syntax.tdl'...
'lexrules.tdl'... 'auxverbs.tdl'... 'letypes.tdl'... 'semrels.tdl'... 'lkb/inflr.tdl'...
'lexicon.tdl'... 'constructions.tdl'... 'lexrinst.tdl'... 'parse-nodes.tdl'...
'roots.tdl'... postloading 'pet/qc-traditional.tdl'...
reading morphology entries from 'pet/full.voc': 17917 entries.
finished parsing - 0 syntax errors, 52021 lines in 1.78 s
processing type constraints (7188 types and 6991 instances):
- expanding templates: 0 template instantiations
- type hierarchy (leaf types [5552], bitcodes, glbs [893], recomputing)
- building dag representation
- computing appropriateness
- applying appropriateness constraints for types
- delta expansion for types
- full type expansion
- shrinking (209729 total nodes, 83178 removed)
- partitioning hierarchy (23 partitions)
- processing instances (expanding --- shrinking [23550/300716])
dumping grammar (symbols 285k, hierarchy 190k, rules & lexicon 350k, types 1160k,
                 instances 677k)
finished conversion - output generated in 10.1 s
```

Configuration

Configuration is done in a file called `flop.settings` containing various switches. On startup, the preprocessor looks for this file in the current directory, and in the subdirectory `pet/`. The lexical syntax of the file is similiar to *TDL* (internally, the file is read using the same module that is used for reading the grammar input files). I will give a specification of the syntax, and an explanation of the possible settings.

Configuration file syntax

config-file → [*setting-list*]
setting-list → *setting* [*setting-list*]
setting → *setting-name* '`:=`' *value-list* '`.`'
setting-name → IDENTIFIER
value-list → *value* [*value-list*]
value → IDENTIFIER | STRING

Settings

preload-files List of file names. These files will be loaded before starting to read the main script file. This is useful to load files containing definitions of *TDL* builtin types that are not known in PET.

postload-files List of file names. These files will be loaded after finishing reading the main script file. This is useful to load files containing definitions that should override previous definitions (similiar to the `lkbpatches.tdl` file for the LKB).

vocabulary-file File name. The contents of this file will be loaded as the full-form lexicon.

version-file File name. This file will be read (using the *TDL*-lexer), and the value read right after encountering the value of the setting *version-string* will be considered the version of the grammar. This allows to propagate the version of the grammar to the following processing stages (cf. *grammar-info*).

version-string String. When reading the version file, that string is the cue for identifying the version of the grammar.

grammar-info Type name. The preprocessor will create an instance with this name, and put information about the grammar (like the version, the number of instances etc.) into this instance. This allows for easy propagation of this information to further processing stages.

rule-status-values List of status values. All instances with one of these status values will be considered a rule of the grammar.

lexicon-status-values List of status values. All instances with one of these status values will be considered a lexical entry.

rule-args-path Feature structure path. The argument of a rule is found under this path in a feature structure.

keyarg-marker-path Feature structure path. The key-daughter is marked with *true-type* under this path (relative to *rule-args-path*).

true-type Type name. This type is used to denote *true*.

head-dtr-path Feature structure path. The head-daughter of a rule can be found under this path.

orth-path Feature structure path. The orthographemic information for a lexical entry is found under this path.

dont-expand List of instance names. These instances are ignored in appropriateness computation, expansion, etc.

sem-attr Attribute name. Used to remove semantic information from feature structures for the command line option *no-semantics*.

output-style Name. If *stefan* is provided as value, the output style of the messages generated by flop will is adapted to match Stefan Mueller's taste.

An Example

```
;;; flop.settings
;;;
;;; this file contains settings for FLOP

;; list of files to load before everything else
preload-files := "".

;; list of files to load after everything else
postload-files := "pet/qc-traditional".

;; file containing the (full form) lexicon
vocabulary-file := "pet/full".

;; file that contains version information
version-file := "Version.lisp".

;; this file is lexed according to TDL syntax, then the string following
;; 'version_string' is taken as the version of the grammar
version-string := "*grammar-version*".

;; name of type to put info about grammar into (if at all)
```

```
grammar-info := grammar_info.

;; list of status values that mark rules
rule-status-values := rule.

;; list of status values that mark lexicon entries
lexicon-status-values := lex-entry.

;; path to the list of arguments in a rule
rule-args-path := ARGS.

;; path to marker for key argument of a rule
keyarg-marker-path := KEY-ARG.

;; type that represents 'true' (for marking the key argument)
true-type := +.

;; path to the head daughter of a rule
head-dtr-path := HEAD-DTR.

;; path to orthography information
orth-path := STEM.

;; list of instances to ignore for appropriateness, expansion etc.
dont-expand := qc_paths.

;; uncomment to enable customized output style for Stefan Mueller
;; output-style := stefan.
```

Command Line Switches

Beside the name of the main *TDL*-file to process, the preprocessor accepts a number of command line switches.

-pre Do only syntactic preprocessing.

-no-expand Do not perform expansion

-expand-all-instances Expand all instances, including lexical instances.

-full-expansion Disable partial expansion (see Section 4.5).

-no-shrink Disable unfilling (see Section 4.5).

-minimal Compute feature layouts for the minimal fixed arity encoding (see Section 4.6.4).

-propagate-status For PAGE compatibility: Propagate status values along the type hierarchy.

-no-semantics Remove all information in feature structures under the attribute specified by the setting *sem-attr*.

-verbose=n Make flop more verbose.

The Parser

Configuration

Configuration of the parser is done in a file called `cheap.settings`. On startup, the parser looks for this file in the current directory, and in the subdirectory `pet/`. The lexical syntax is the same as for the `flop.settings` file, see above.

Settings

grammar-info Type name. Name of type that contains grammar information. Cf. the corresponding setting for flop.

start-symbols List of type names. A valid parse has to be compatible with one of the specified types.

deleted-daughters List of attribute names. Values under these attributes are not passed from the mother to the daughter in parsing (restrictor, see Section 3.3).

affixation-path Feature structure path. Where to unify the affix when building a full form.

bi-cons-name Type name. Name of the builtin *cons* type.

qc-structure Type name. Name of the type containing the quick check structure.

stop-characters String. Stop characters for the tokenizer.

lex-entries-can-fail Assertion. If stated, lexical entries are allowed to fail in expansion.

translate-iso-chars Assertion. Translate ISO-8859-1 characters to ASCII.

chart-dependencies List of pairs of feature structure paths. Used to specify mutual selection for *chart manipulation*, see Section 3.3.

generic-les List of type names. Generic lexical entries for unknown words: for each unknown word in the input all generic entries are postulated.

generic-le-suffixes List of pairs of type names and strings. Lexical entres that require a certain suffix are only postulated if the input form has the suffix.

default-gen-le-priority Integer priority value. Scoring for generic items is based on the default priority specified here (typically low).

posmapping List of triples of strings, integer priority values and type names. Priorities for generic lexical entries are adjusted on the basis of POS information that may be available for the unknown word. If the input word has one more more POS tags associated to it, these are looked up in this table, which is a list of triples (tag, score, gle) where gle is the name of one of the generic items in *generic-les*. For each generic item, the score is adjusted to the first match of one of the tags associated with the unknown word in the mapping table.

default-rule-priority Integer priority value. Default priority for rules.

rule-priorities List of pairs of rule names and integer priority values. Specifies priorities for rules.

default-le-priority Integer priority value. Default priority for lexical entries.

unlikely-le-types List of pairs of lexical type names and integer priority values. Specifies (typically low) priorities for lexical entries.

likely-le-types List of pairs of lexical type names and integer priority values. Specifies (typically high) priorities for lexical entries.

An Example

```
;;; cheap.settings
;;;
;;; this file contains settings for CHEAP

;; name of type to get info about the grammar (if at all)
grammar-info := grammar_info.

;; types of a valid parse
start-symbols := root_strict root_phr root_lex root_subord root_conj.

;; names of attributes not to pass from daughter to mother in parsing
deleted-daughters := ARGS HEAD-DTR NON-HEAD-DTR LCONJ-DTR RCONJ-DTR.

;; prefix of list-valued path where to unify the affix
affixation-path := "ARGS.FIRST".

;; name of the builtin cons type
bi-cons-name := "*cons*".

;; name of type containing quick check structure
qc-structure := qc_paths.

;; list of stop characters for the tokenizer, default is "\t?!.:;,()-+*$\n"
stop-characters := "\t?!.:;,()+*$\n".

;; allow creation of lexical entries to fail
lex-entries-can-fail.

;; translate german umlaut and sz
translate-iso-chars.
```

```
;; chart manipulation for seperable prefixes
;; contains a list of pairs of paths
chart-dependencies := "SYNSEM.LOC.CAT.HEAD.SP-FORM"
                      "SYNSEM.LOC.CAT.SP.FIRST.LOC.CAT.HEAD.SP-FORM".

;; generic lexical entries for unknown words
generic-les := generic_unerg_verb_bse generic_unerg_verb_pres3sg
               generic_unerg_verb_presn3sg generic_unerg_verb_past
               generic_unerg_verb_psp generic_unerg_verb_prp
               generic_trans_verb_bse generic_trans_verb_pres3sg
               generic_trans_verb_presn3sg generic_trans_verb_past
               generic_trans_verb_psp generic_trans_verb_prp
               generic_sg_noun generic_pl_noun
               genericname generic_adj generic_adverb.

;; suffixes required by generic lexical entries
generic-le-suffixes := generic_unerg_verb_pres3sg "S"
                       generic_unerg_verb_past "ED"
                       generic_unerg_verb_psp "ED"
                       generic_unerg_verb_prp "ING"
                       generic_trans_verb_pres3sg "S"
                       generic_trans_verb_past "ED"
                       generic_trans_verb_psp "ED"
                       generic_trans_verb_prp "ING" generic_pl_noun "S".

;; default priority for generic lexical entries
default-gen-le-priority := 10.

;; priorities for generic entries from POS information
posmapping := verb 500 generic_unerg_verb_pres3sg
              tverb 500 generic_trans_verb_pres3sg
              pnoun 500 genericname.


;; default priority for rules
default-rule-priority := 500.

;; scoring for grammar rules
rule-priorities := extrasubj_f 300
                   freerel 100
                   noun_n_cmpnd 300
                   top_coord_np 700
                   hcomp 800
                   bare_np 300
                   bare_vger 300
                   numadj_np 100
                   dative_lr 400.

default-le-priority := 400.

unlikely-le-types := letter_name_le 100
                     n_freerel_pro_le 200
                     n_freerel_pro_adv_le 210
                     n_proper_abb_le 200
                     n_deictic_pro_le 200.

likely-le-types := conj_complex_le 800
                   adv_disc_le 800
                   comp_to_nonprop_le 800
                   p_subconj_inf_le 800
                   v_empty_prep*_trans_nosubj_le 800.
```

Command Line Switches

The parser accepts the following command line switches:

-tsdb Connect to [incr tsdb()].

-one-solution Run in best-first mode.

-limit=n Limit the number of passive edges in the chart.

-no-filter Disable the static rule filter.

-qc=n Use only the first n quick check paths.

-compute-qc Compute quick check paths.

-key=n Select key mode. 0 is key-driven, 1 is strict left to right, 2 is strict left to right, and 3 is head-driven.

-rulestats Enable reporting of rule statistics to [incr tsdb()]. Useful for determining key daughters (see Section 3.3).

-no-hyper Disable hyper-active parsing.

-no-derivation Do not output derivations.

-no-chart-man Disable chart manipulation (see Section 3.3).

-default-les Use generic lexical entries.

-no-shrink-mem Do not attempt to shrink the process size after parsing huge items.

-verbose=n Make cheap more verbose.

D. Virtual Appendix

Besides the printed appendix, this thesis provides a virtual appendix that gives you access to the raw results of the discussed experiments, in the form of [incr tsdb()] profiles, gprof profiles, or collections of parser logfiles that can be processed using standard Unix text processing tools. The virtual appendix also provides the sources of the PET system. You can access the virtual appendix under the following address:

```
http://www.coli.uni-sb.de/~uc/thesis/
```

I provide the raw result data of the experiments so that you can study aspects of the data that I could not include in my discussion, and so that you can compare my results with that of other systems. Access to the sources allows you to reproduce my experiments if you wish, and also opens the possibility for you to experiment with other processing techniques in the context of the PET system.

Bibliography

Hassan Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction*. MIT Press, Cambridge, MA, 1991.

Hassan Aït-Kaci, Robert Boyer, Patrick Lincoln, and Roger Nasr. Efficient Implementation of Lattice Operations. *ACM Transactions on Programming Languages and Systems*, 11(1):115–146, 1989.

Hassan Aït-Kaci and Roberto Di Cosmo. Compiling Order-Sorted Feature Term Unification. Technical report, Digital Paris Research Laboratory, December 1993. PRL Technical Note 7.

Hassan Aït-Kaci, Andreas Podelski, and Seth Copen Goldstein. Order-Sorted Feature Theory Unification. Technical Report 32, Digital Equipment Corporation, DEC Paris Research Laboratory, France, May 1993. Also in Proceedings of the International Symposium on Logic Programming, Oct. 1993, MIT Press.

Gosse Bouma and Gertjan van Noord. Head-driven Parsing for Lexicalist Grammars. Experimental Results. In *Proceedings of the 6th Conference of the European Chapter of the ACL*, pages 71–80, Utrecht, The Netherlands, 1993.

R. S. Boyer and J. S. Moore. The sharing of structure in theorem-proving programs. In *Machine Intelligence 7*, pages 101–116, New York, New York, 1972.

Joan Bresnan and Ronald M. Kaplan. Lexical-Functional Grammar: a formal system for grammatical representation. In Joan Bresnan, editor, *The Mental Representation of Grammatical Relations*. MIT Press, Cambridge, MA, 1982.

John C. Brown and Suresh Manandhar. An Abstract Machine for Fast Parsing of Typed Feature Structure Grammars. In Stephan Diel and Peter Sestoft, editors, *Proceedings of the Workshop on Principles of Abstract Machines*, number 05/98. Fachbereich Informatik, Universität des Saarlandes, Saarbrücken, Germany, 1998.

John C. Brown and Suresh Manandhar. Compilation versus abstract machines for fast parsing of typed feature structure grammars. *Future Generation Computer Systems*, 16 (7):771–791, 2000.

Ulrich Callmeier. PET — A Platform for Experimentation with Efficient HPSG Processing Techniques. *Natural Language Engineering*, 6 (1) (Special Issue on Efficient Processing with HPSG):99–108, 2000.

Ulrich Callmeier. Preprocessing and Encoding Techniques in PET. In Oepen et al. (2001). Forthcoming.

Bob Carpenter. ALE – the attribute logic engine: User's guide. Technical report, Laboratory for Computational Linguistics, Philosophy Department, Carnegie Mellon University, Pittsburgh, PA, December 1992a.

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

Bob Carpenter and Gerald Penn. Compiling typed attribute-value logic grammars. In *Recent Advances in Parsing Technology*, pages 145–168. Kluwer, 1996.

Bob Carpenter and Yan Qu. An Abstract Machine for Attribute-Value Logics. In *Proceedings of the 4th International Workshop on Parsing Technologies*, Prague, Czech Republik, 1995.

John Carroll. Relating Complexity to Practical Performance in Parsing with Wide-Coverage Unification Grammars. In *Proceedings of the 32nd Meeting of the Association for Computational Linguistics*, pages 287–294, Las Cruces, NM, 1994.

L.-V. Ciortuz. Scaling up the abstract machine for unification of OSF-terms to do head-corner parsing with large-scale typed unification grammars. In *Proceedings of the ESSLLI 2000 Workshop on Linguistic Theory and Grammar Implementation*, pages 57–80, Birmingham, UK, August 14–18, 2000.

Ann Copestake. The ACQUILEX LKB. Representation Issues in Semi-Automatic Acquisition of Large Lexicons. In *Proceedings of the 3rd ACL Conference on Applied Natural Language Processing*, pages 88–96, Trento, Italy, 1992.

Ann Copestake. The (new) LKB system. CSLI, Stanford University, CA. `http://www-csli.stanford.edu/~aac/doc5-2.pdf`, 1999.

Ann Copestake. Appendix: Definitions of typed feature structures. *Natural Language Engineering*, 6 (1) (Special Issue on Efficient Processing with HPSG), 2000.

Ann Copestake and Daniel P. Flickinger. An open-source grammar development environment and broad-coverage English grammar using HPSG. In *Proceedings of the Second Linguistic Resources and Evaluation Conference*, pages 591–600, Athens, Greece, 2000.

Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.

Mary Dalrymple, Ronald M. Kaplan, John T. Maxwell III, and Annie Zaenen, editors. *Formal Issues in Lexical-Functional Grammar*. CSLI Lecture Notes. Cambridge University Press, Stanford, CA, 1995.

Karel Driesen. *Software and Hardware Techniques for Efficient Polymorphic Calls*. PhD thesis, Department of Computer Science, University of California, Santa Barbara, CA, 1999. Published as TRCS99-24.

Martin C. Emele. Unification with lazy non-redundant copying. In *Proceedings of the 29th Meeting of the Association for Computational Linguistics*, pages 323–330, Berkeley, CA, 1991.

Martin C. Emele and Rémi Zajac. A fix-point semantics for feature type systems. In *Proceedings of the 2nd International Workshop on Conditional and Typed Rewriting, CTRS'90*, pages 383–388, Montreal, Canada, 1990a.

Martin C. Emele and Rémi Zajac. Typed unification grammars. In *Proceedings of the 13th International Conference on Computational Linguistics*, pages 293–298, Helsinki, Finland, 1990b.

Gregor Erbach. A Bottom-Up Algorithm for Parsing and Generation. CLAUS Report 5, Computational Linguistics at the University of the Saarland, Saarbrücken, Germany, February 1991a.

Gregor Erbach. A Flexible Parser for a Linguistic Development Environment. In Otthein Herzog and Claus-Rainer Rollinger, editors, *Text Understanding in LILOG*, pages 74–87. Springer, Berlin, Germany, 1991b.

Gregor Erbach. An Environment for Experimenting with Parsing Strategies. In John Mylopoulos and Ray Reiter, editors, *Proceedings of IJCAI 1991*, pages 931–937, San Mateo, CA, 1991c. Morgan Kaufmann Publishers.

Gregor Erbach. ProFIT: Prolog with features, inheritance and templates. CLAUS Report 42, Computational Linguistics at the University of the Saarland, Saarbrücken, Germany, July 1994.

Daniel P. Flickinger. On Building a More Efficient Grammar by Exploiting Types. *Natural Language Engineering*, 6 (1) (Special Issue on Efficient Processing with HPSG):15–28, 2000.

Daniel P. Flickinger, Ann Copestake, and Ivan A. Sag. HPSG Analysis of English. In Wahlster (2000), pages 255–264.

Daniel P. Flickinger, Stephan Oepen, Jun ichi Tsujii, and Hans Uszkoreit, editors. *Journal of Natural Language Engineering. Special Issue on Efficient Processing with HPSG: Methods, Systems, Evaluation*. Cambridge University Press, Cambridge, UK, 2000b.

Daniel P. Flickinger, Carl Pollard, and Thomas Wasow. Structure-Sharing in Lexical Representation. In *Proceedings of the 23rd Meeting of the Association for Computational Linguistics*, pages 262–267, Chicago, IL, 1985.

Daniel P. Flickinger and Ivan A. Sag. Linguistic Grammars Online. A Multi-Purpose Broad-Coverage Computational Grammar of English. In *CSLI Bulletin 1999*, pages 64–68, Stanford, CA, 1998. CSLI Publications.

Gerald Gazdar, Ewan Klein, Geoffrey K. Pullum, and Ivan A. Sag. *Generalized Phrase Structure Grammar*. Blackwell Publishing and Harvard University Press, Oxford, England and Cambridge, MA, 1985.

Dale Gerdemann. Term Encoding of Typed Feature Structures. In *Proceedings of the 4th International Workshop on Parsing Technologies*, pages 89–97, Prague, Czech Republik, 1995.

Kurt Godden. Lazy Unification. In *Proceedings of the 28th Meeting of the Association for Computational Linguistics*, pages 180–187, Pittsburgh, PA, 1990.

Thilo Götz. A normal form for typed feature structures. Magisterarbeit, Universität Tübingen, Tübingen, Germany, 1993.

Barbara J. Grosz, Karen Sparck Jones, and B. L. Webber, editors. *Readings in Natural Language Processing*. M. Kaufmann, Los Altos, CA, 1986.

Lauri Karttunen. D-PATR: A Development Environment for Unification-Based Grammars. Technical Report CSLI-86-61, Center for the Study of Language and Information, Stanford University, 1986.

Lauri Karttunen and Martin Kay. Structure Sharing with Binary Trees. In *Proceedings of the 23rd Meeting of the Association for Computational Linguistics*, pages 133–136A, Chicago, IL, 1985.

Martin Kay. Algorithm Schemata and Data Structures in Syntactic Processing. In Grosz et al. (1986), pages 35–70. Originally published as a Xerox PARC technical report CSL-80-12, 1980.

Martin Kay. Head-driven Parsing. In *Proceedings of the 1st International Workshop on Parsing Technologies*, pages 52–62, Pittsburgh, PA, 1989.

Bernd Kiefer and Hans-Ulrich Krieger. A context-free approximation of Head-driven Phrase Structure Grammar. In *Proceedings of the 6th International Workshop on Parsing Technologies*, pages 135–146, Trento, Italy, 2000.

Bernd Kiefer, Hans-Ulrich Krieger, John Carroll, and Robert Malouf. A Bag of Useful Techniques for Efficient and Robust Parsing. In *Proceedings of the 37th Meeting of the Association for Computational Linguistics*, pages 473–480, College Park, MD, 1999.

Bernd Kiefer, Hans-Ulrich Krieger, and Mark-Jan Nederhof. Efficient and Robust Parsing of Word Hypotheses Graphs. In Wahlster (2000), pages 261–296.

Kiyoshi Kogure. Strategic Lazy Incremental Copy Graph Unification. In *Proceedings of the 13th International Conference on Computational Linguistics*, pages 223–228, Helsinki, Finland, 1990.

Hans-Ulrich Krieger and Ulrich Schäfer. $\mathcal{TDL}$— A Type Description Language for Constraint-Based Grammars. In *Proceedings of the 15th International Conference on Computational Linguistics*, pages 893–899, Kyoto, Japan, 1994a.

Hans-Ulrich Krieger and Ulrich Schäfer. $\mathcal{TDL}$— A Type Description Language for HPSG. Part 2: User Guide. Document D-94-14, German Research Center for Artificial Intelligence (DFKI), Saarbrücken, Germany, 1994b.

Hans-Ulrich Krieger and Ulrich Schäfer. Efficient Parameterizable Type Expansion for Typed Feature Formalisms. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, pages 1428–1434, San Francisco, CA, 1995.

Alfredo M. Maeda, Jin-Ichi Aoe, and Hideto Tomabechi. Signature-check Based Unification Filter. *Software – Practice and Experience*, 24(7):603–622, 1994.

Takaki Makino, Kentaro Torisawa, and Jun-Ichi Tsujii. LiLFeS — Practical Programming Language For Typed Feature Structures. In *Proceedings of the Natural Language Processing Pacific Rim Symposium*, 1997.

Takaki Makino, Minoru Yoshida, Kentaro Torisawa, and Jun-ichi Tsujii. LiLFeS — Towards a Practical HPSG Parser. In *Proceedings of the 17th International Conference on*

Computational Linguistics and the 36th Annual Meeting of the Association for Computational Linguistics, pages 807–11, Montreal, Canada, 1998.

Robert Malouf, John Carroll, and Ann Copestake. Efficient Feature Structure Operations without Compilation. *Natural Language Engineering*, 6 (1) (Special Issue on Efficient Processing with HPSG):29–46, 2000.

Kurt Mehlhorn. *Data structures and Algorithms 2: Graph Algorithms and NP-Completeness*, volume 2. Springer, Berlin;Heidelberg;New York, 1984.

Yusuke Miyao, Takaki Makino, Kentaro Torisawa, and J. Tsujii. The LiLFeS Abstract Machine and its Evaluation with the LinGO Grammar. *Natural Language Engineering*, 6 (1) (Special Issue on Efficient Processing with HPSG):47–61, 2000.

Stefan Müller. *Deutsche Syntax deklarativ. Head-Driven Phrase Structure Grammar für das Deutsche*. Number 394 in Linguistische Arbeiten. Max Niemeyer Verlag, Tübingen, 1999. `http://www.dfki.de/~stefan/Pub/e_hpsg.html`.

Stefan Müller and Walter Kasper. HPSG Analysis of German. In Wahlster (2000), pages 238–253.

Stephan Oepen. *Competence and Performance Profiling for Constraint-based Processing*. PhD thesis, Computational Linguistics, Saarland University, 2001.

Stephan Oepen and Ulrich Callmeier. Measure for measure: Parser Cross-Fertilization. Towards increased component comparability and exchange. In *Proceedings of the 6th International Workshop on Parsing Technologies*, pages 183–194, Trento, Italy, 2000.

Stephan Oepen and John Carroll. Ambiguity Packing in Constraint-based Parsing. Practical Results. In *Proceedings of the 1st Conference of the North American Chapter of the ACL*, pages 162–169, Seattle, WA, 2000a.

Stephan Oepen and John Carroll. Performance Profiling for Parser Engineering. *Natural Language Engineering*, 6 (1) (Special Issue on Efficient Processing with HPSG):81–97, 2000b.

Stephan Oepen and Daniel P. Flickinger. Towards Systematic Grammar Profiling. Test Suite Technology Ten Years After. *Journal of Computer Speech and Language*, 12 (4) (Special Issue on Evaluation):411–436, 1998.

Stephan Oepen, Daniel P. Flickinger, Jun-ichi Tsujii, and Hans Uszkoreit, editors. *Collaborative Language Engineering. A Case Study in Efficient Grammar-based Processing*. CSLI Publications, Stanford, CA, 2001. Forthcoming.

Stephan Oepen, Daniel P. Flickinger, Hans Uszkoreit, and Jun-ichi Tsujii. Introduction to this Special Issue. *Natural Language Engineering*, 6 (1) (Special Issue on Efficient Processing with HPSG), 2000.

Stephan Oepen, Klaus Netter, and Judith Klein. TSNLP — Test Suites for Natural Language Processing. In John Nerbonne, editor, *Linguistic Databases*, pages 13–36. CSLI Publications, Stanford, CA, 1997.

Gerald Penn. *The Algebraic Structure of Attributed Type Signatures*. PhD thesis, School of Computer Science, Carnegie Mellon University, 2000. `http://www.cs.cmu.edu/~gpenn/cards/thesis.html`.

Fernando C. N. Pereira. A Structure-Sharing Representation for Unification-Based Grammar Formalisms. In *Proceedings of the 23rd Meeting of the Association for Computational Linguistics*, pages 137–144, Chicago, IL, 1985.

Carl Pollard and Ivan A. Sag. *Information-Based Syntax and Semantics. Volume 1: Fundamentals*. CSLI Lecture Notes # 13. Center for the Study of Language and Information, Chicago, IL and Stanford, CA, 1987. Distributed by The University of Chicago Press.

Carl Pollard and Ivan A. Sag. *Head-Driven Phrase Structure Grammar*. Studies in Contemporary Linguistics. The Univeristy of Chicago Press and CSLI Publications, Chicago, IL and Stanford, CA, 1994.

Derek Proudian and Carl Pollard. Parsing Head-Driven Phrase Structure Grammar. In *Proceedings of the 23rd Meeting of the Association for Computational Linguistics*, pages 167–171, Chicago, IL, 1985.

Yan Qu. An Abstract Machine for Typed Feature Structure Grammar Theories. Master's thesis, Carnegie Mellon University, Pittsburgh, PA, 1994.

Christian Schulte. Comparing Trailing and Copying for Constraint Programming. In Danny De Schreye, editor, *Proceedings of the Sixteenth International Conference on Logic Programming*, pages 275–289, Las Cruces, NM, USA, November 1999. The MIT Press.

Stuart M. Shieber. Using Restriction to Extend Parsing Algorithms for Complex Feature-Based Formalisms. In *Proceedings of the 23rd Meeting of the Association for Computational Linguistics*, pages 145–152, Chicago, IL, 1985.

Stuart M. Shieber. *An Introduction to Unification-Based Approaches to Grammar*. CSLI Lecture Notes. Center for the Study of Language and Information, Stanford, CA, 1986.

Melanie Siegel. HPSG Analysis of Japanese. In Wahlster (2000), pages 265–280.

Yuka Tateisi, Kentaro Torisawa, Yusuke Miyao, and Jun ichi Tsujii. Translating the XTAG English grammar to HPSG. In *Proceedings of the 4th Workshop on Tree-adjoining Grammars and Related Frameworks (TAG+)*, pages 172–175, Philadelphia, PA, 1998.

Hideto Tomabechi. Quasi-Destructive Graph Unification. In *Proceedings of the 29th Meeting of the Association for Computational Linguistics*, pages 315–322, Berkeley, CA, 1991.

Hideto Tomabechi. Quasi-Destructive Graph Unification with Structure-Sharing. In *Proceedings of the 14th International Conference on Computational Linguistics*, pages 440–446, Nantes, France, 1992.

Hideto Tomabechi. Design of Efficient Unification for Natural Language. *Journal of Natural Language Processing*, 2(2):23–58, 1995.

Kentaro Torisawa, Kenji Nishida, Yusuke Miyao, and J. Tsujii. An HPSG Parser with CFG Filtering. *Natural Language Engineering*, 6 (1) (Special Issue on Efficient Processing with HPSG):63–80, 2000.

Kentaro Torisawa and Junichi Tsujii. Computing phrasal signs in HPSG prior to parsing. In *Proceedings of the 16th International Conference on Computational Linguistics*, pages 949 – 955, Kopenhagen, Denmark, 1996.

Hans Uszkoreit. Strategies for Adding Control Information to Declarative Grammars. In *Proceedings of the 29th Meeting of the Association for Computational Linguistics*, pages 237–245, Berkeley, CA, 1991.

Hans Uszkoreit, Rolf Backofen, Stephan Busemann, Abdel Kader Diagne, Elizabeth A. Hinkelman, Walter Kasper, Bernd Kiefer, Hans-Ulrich Krieger, Klaus Netter, Günter Neumann, Stephan Oepen, and Stephen P. Spackman. DISCO — An HPSG-based NLP System and its Application for Appointment Scheduling. In *Proceedings of the 15th International Conference on Computational Linguistics*, pages 436–440, Kyoto, Japan, 1994. A version of this paper is available as DFKI Research Report RR-94-38.

Marcel P. van Lohuizen. Memory-Efficient and Thread-Safe Quasi-Destructive Graph Unification. In *Proceedings of the 38th Meeting of the Association for Computational Linguistics*, pages 352–359, Hong Kong, China, October 2000.

Marcel P. van Lohuizen. Efficient and Thread-Safe Unification with LinGO. In Oepen et al. (2001). Forthcoming.

Gertjan van Noord. An Efficient Implementation of the Head-Corner Parser. *Computational Linguistics*, 23 (3):425–456, 1997.

Peter Van Roy. 1983–1993: The Wonder Years of Sequential Prolog Implementation. *Journal of Logic Programming*, 19/20:385–441, May/July 1994. Special Issue: Ten Years of Logic Programming.

Wolfgang Wahlster, editor. *Verbmobil: Foundations of Speech-to-Speech Translation*. Artificial Intelligence. Springer-Verlag, Berlin, Heidelberg, New York, 2000.

David H. D. Warren. *Applied Logic – its use and implementation as a programming tool*. PhD thesis, University of Edinburgh, Edinburgh, Scotland, 1977.

David H. D. Warren. An abstract Prolog instruction set. Technical Note 309, SRI International, October 1983a.

David H. D. Warren. Logarithmic access arrays for Prolog. Unpublished program, 1983b.

S. Wintner, N. Francez, and E. Gabrilovich. AMALIA – a unified platform for parsing and generation. In Ruslan Mitkov, Nicolas Nicolov, and Nikolai Nikolov, editors, *Recent Advances in Natural Language Processing*, pages 135–142, Tzigov Chark, Bulgaria, September 1997.

Shalom Wintner. *An Abstract Machine for Unification Grammars with Applications to an HPSG Grammar for Hebrew*. PhD thesis, Technion, Israel Institute of Technology, Haifa, Israel, 1997.

Shuly Wintner and Nissim Francez. An Abstract Machine for Typed Feature Structures. In *Proceedings of the 5th Workshop on Natural Language Understanding and Logic Programming*, pages 205–220, Lisbon, Portugal, 1995.

Shuly Wintner and Nissim Francez. Efficient Implementation of Unification-Based Grammars. *Journal of Language and Computation*, 1(1):53–92, April 1999.

David A. Wroblewski. Nondestructive Graph Unification. In *Proceedings of the 6th National Conference on Artificial Intelligence*, pages 582–587, Seattle, WA, 1987.

Index

[incr tsdb()], 3, 30, 32, 35, *36*

Aït-Kaci et al. (1993), 22
abstract machines, 22–24, 50
ALE, 23
AMALIA, 22, 50
Aït-Kaci and Di Cosmo (1993), 22, 50
Aït-Kaci et al. (1989), 28, 29
Aït-Kaci (1991), 22, 34, 50

Bouma and van Noord (1993), 32
Boyer and Moore (1972), 6
Bresnan and Kaplan (1982), 1
Brown and Manandhar (1998), 22
Brown and Manandhar (2000), 22

cali, 27, 38, 69
Callmeier (2000), 2
Callmeier (2001), 2
Carpenter and Penn (1996), 23
Carpenter and Qu (1995), 22, 23, 47
Carpenter (1992a), 23
Carpenter (1992b), 1, 3
Carroll (1994), 2, 35, 61
CHIC, 22, 38, 50
Ciortuz (2000), 22, 47, 50
compilation, 22–24, 63
Copestake and Flickinger (2000), 2, 3, 38
Copestake (1992), 2, 3, 25
Copestake (1999), 25, 65
Copestake (2000), 1, 3, 47
copying
- early, *8*, *18*, 19, 39
- lazy, 5, 17
- over, *8*, 39
- redundant, 15, *16*, 17, 19, 39

Cormen et al. (1990), 31

Dalrymple et al. (1995), 1
Driesen (1999), 50

Emele and Zajac (1990a), 17
Emele and Zajac (1990b), 18
Emele (1991), 6, 8, 9, 17, 50
environment, 6, 55
Erbach (1991a), 30
Erbach (1991b), 21, 31, 45
Erbach (1991c), 61
Erbach (1994), 23
expansion, 47–49
- partial, 47

failure frequency, 16, 45
Flickinger and Sag (1998), 2
Flickinger et al. (1985), 61
Flickinger et al. (2000a), 3, 38
Flickinger et al. (2000b), 2, 38
Flickinger (2000), 44, 49
functions
- AssignCode, 28, 29
- clock(3), 39
- force-delayed-copy, 14
- operator[], 31
- unify1, 9, 11, 12, 14, 19, 26, 41
- unify2, 9–14, 16, 20, 26, 41, 52
- unify3, 12–14, 26, 41

Gazdar et al. (1985), 1
Gerdemann (1995), 27, 48
Godden (1990), 6, 8, 9, 14, 15, 17, 22
gprof, 42, 56
Götz (1993), 27, 47, 48

hardware, 39

Karttunen and Kay (1985), 5, 6, 21, 22
Karttunen (1986), 7, 8, 21
Kay (1986), 30
Kay (1989), 32
Kiefer and Krieger (2000), 22, 45
Kiefer et al. (1999), 6, 9, 16, 21, 30–33, 45
Kiefer et al. (2000), 31
Kogure (1990), 6, 8, 9, 15, 17, 21, 22, 40, 45, 50
Krieger and Schäfer (1994a), 1, 3, 27
Krieger and Schäfer (1994b), 65
Krieger and Schäfer (1995), 27

lazy evaluation, 14, 16
LiLFeS, 22, 23, 38, 50, 63
Lisp, 2, 33
LKB, iii, 2–4, 25, 28, 30, 32, 33, 37, 38, 41, 46, 47, 49, 57–59, 61, 65, 76

Maeda et al. (1994), 21, 45
Makino et al. (1997), 22
Makino et al. (1998), 2, 22, 50
Malouf et al. (2000), 5, 6, 8, 9, 12, 16, 19–22, 26, 31, 40, 41, 45–48, 50, 52
Mehlhorn (1984), 29
memory management, 33
Miyao et al. (2000), 22, 24, 63
money, 1
Müller and Kasper (2000), 3, 38
Müller (1999), 3, 38

Oepen and Callmeier (2000), 2, 3, 31–33, 36, 37
Oepen and Carroll (2000a), 33, 61
Oepen and Carroll (2000b), 3, 32, 33, 36, 54
Oepen and Flickinger (1998), 3, 36
Oepen et al. (1997), 36
Oepen et al. (2000), 3, 69
Oepen et al. (2001), 2, 38
Oepen (2001), 3, 36, 57

PAGE, 2, 3, 25, 28, 30–32, 38, 47, 49, 61, 62, 78
parsing
    hyper-active, 32, 54
    key-driven, 31
Penn (2000), 1, 22–24, 28, 63
Pereira (1985), 5, 6, 8, 17, 21, 22
Pollard and Sag (1987), 1
Pollard and Sag (1994), 1
profiling
    competence & performance, 36
    execution time, 42–44
ProFIT, 23
Prolog, 23–24, 50
Proudian and Pollard (1985), 61

Qu (1994), 22
quick check, 6, 16, 21, 32, 45–47

representation
    of feature structures, 5, 49–54
restrictor, 32
rule filter, 32
rule indexation, 43

Schulte (1999), 56
sharing
    data-structure, 5
    feature-structure, 5
    structure, 5–7, 15, 17, 19
    subgraph, 5, 19
Shieber (1985), 32
Shieber (1986), 1
Siegel (2000), 3, 38
skeleton, 6, 55
space
    wasted, 52

Tateisi et al. (1998), 2
Tomabechi (1991), 6–9, 18, 20–23, 26, 33, 40, 41, 50
Tomabechi (1992), 5, 8, 19–22
Tomabechi (1995), 8, 16, 18
Torisawa and Tsujii (1996), 2

Torisawa et al. (2000), 22, 45
trailing, 55

unfilling, 48–49
Uszkoreit et al. (1994), 1
Uszkoreit (1991), 16, 21, 45

van Lohuizen (2000), 18, 20–22, 26
van Lohuizen (2001), 20, 26, 27, 69
Van Roy (1994), 22
van Noord (1997), 32

WAM, 22, 34, 49, 50
Warren (1977), 6
Warren (1983a), 22, 34, 50
Warren (1983b), 7
Wintner and Francez (1995), 22
Wintner and Francez (1999), 22
Wintner et al. (1997), 22
Wintner (1997), 22, 47, 50
Wroblewski (1987), i, ii, 3, 7–11, 13, 16, 18, 20, 21, 26, 40, 50, 52