Abstract

We consider two alternatives for memory management in typed-feature-structure-based parsers by identifying structural properties of grammar signatures that may be of some predictive value in determining the consequences of those alternatives. We define these properties, summarize the results of a number of experiments on artificially constructed signatures with respect to the relative rank of their asymptotic cost at parse-time, and experimentally consider how they impact memory management.

1 Introduction

Memory management deals with organizing the compiled object of a computer program so as to consume less memory for the same amount of work. When the overall memory consumption becomes so large that it swaps out to disk, better memory management can also make the compiled object considerably faster. HPSG parsing, particularly with large grammars such as the English Resource Grammar (Copestake and Flickinger, 2000), has a number of problems with memory consumption. Very often, parsing charts must be pruned or chart-parsing must be terminated early because the overall memory consumption is too great for a grammar developer's desktop computer.

Memory managers must decide how to allocate memory to an application over the course of an execution, detect when an application no longer requires a certain location in memory, and recycle locations that are no longer needed. A central concern to all of these tasks is the size of the blocks of memory that are allocated, monitored and recycled.

Current HPSG parsers do have memory managers — relying on the operating system is simply not an option. The ALE system (Carpenter and Penn, 1996) uses SICStus Prolog's memory manager, and the LKB (Copestake and Flickinger, 2000) uses Allegro Common Lisp's memory manager. PET (Callmeier, 2001) actually comes with a few options, including using pools of fixed-size memory blocks à la C++, a Windows-style virtual memory manager, and a special 2-stack version of the model that Prolog uses. LiLFeS (Makino et al., 1998) has its own memory manager for logic programming with typed feature structures, which at least in early versions of that system, put its performance well behind that of SICStus Prolog (Penn, 2000).

In the context of HPSG parsing, the central memory management question has been whether to (re-)allocate memory for feature structures in blocks that exactly correspond to the arity of their current type (a block consists of an encoding of the type plus n pointers to each of the n appropriate feature values for that type) or to allocate it in blocks that are, on occasion, larger or smaller than what is currently needed. The argument for adding extra space is connected to the subtype polymorphism that is inherent to the logic of typed feature structures. While each type does have a fixed arity of features appropriate to it, that type may be promoted to a subtype, whereupon it may acquire more such features. If extra space is allocated to

the feature structure at the outset, the *frame* that stores the type and pointers to the appropriate feature values does not need to be resized, moved or reallocated. The argument for allocating less space is equally compelling, especially when certain feature values can be inferred from context. There is, in fact, a great experimental evidence that in large practical grammars, it pays even to re-derive certain feature values as needed. Research on this began with Goetz's work on the Troll system (Goetz, 1993), in which he coined the term *unfilling*, and nearly every system for HPSG parsing since then has experimented with some form of this. We will not discuss unfilling more in this paper; for the purposes of this study, one can either leave even more cells empty or tighten the representation up even further, so the same choice that we address here remains present even when unfilling is used.

Most of the previous work on "memory management" in HPSG parsing has focussed on specialized unification algorithms for this task that avoid copying. While these exert a great influence upon the operating conditions of the memory manager, they do not by themselves manage memory, nor do they completely answer the central question posed above: what the size of the allocated frames should be. Lower-level research that directly pertains to that question is far more sparse and what there is is mostly anecdotal. Penn (2000) experimented with what he called a *variable* approach, in which the number of available feature slots was exactly the number of appropriate features to the current type, and a *fixed* approach, in which enough extra space was allocated, as determined by a coarse modularization of the type signature and a graph colouring algorithm, to guarantee that the frame would never need to be relocated. He tested both of these on two grammars: the ALE HPSG grammar (Penn, 1993) 93), in which the fixed approach was slightly better, and a categorial grammar written in typed feature logic for the telephone banking domain from Bell Labs, in which the fixed approach was significantly better. Callmeier (2001) also experimented with a fixed and variable approach, although his description of his fixed approach involved modularization with no graph colouring. He found that the variable approach worked better on the English Resource Grammar, and that the fixed approach worked slightly better on the Japanese Verbmobil grammar. While it is clear from both theses that the authors appreciated that the signatures and the distribution of feature structures over types played a very prominent role in determining which method was better, neither leaves the reader with any indication of what it specifically is about those signatures that would favour either of these approaches.

Our purpose in undertaking the study described here has been to complement this earlier work on real grammars by testing both approaches on a range of analytically formulated signatures with very controlled characteristics. This control allows us to determine some of the various dimensions of a type signature's complexity that influence whether the fixed or variable approach will be more beneficial. Real grammars are still important, as are the corpora on which they are evaluated, because these provide the empirical distributions over these characteristics that determine the weights on these analytic variables as they combine to yield the overall cost of the memory management strategy used. Our belief, however,

is that the study of real grammars was perhaps premature. Prior to the present study, we did not even know what the variables were — hence the oblique timings reported in previous work on this subject.

Some of those variables are very task-specific, actually. For example, chart parsing is $\mathcal{O}(n^{MM}G^2\delta^2)$, and the number of edge accesses is known to be influenced by the edge's position and the parsing control strategy. We focussed on unification. This is important to everyone who uses feature structures. HPSG has the added benefit of a type system for its feature structures, which allows us to do more static analysis and less empirical analysis than in grammar formalisms in which their untyped historical precursors are used.

The potential benefits of this direction of research are twofold. First, it can serve as a guide to grammar writers, so that they may be able to choose more efficient encodings of linguistic constructions in signatures, when several acceptable ones present themselves. Second, it can serve as a guide to system developers, who will be able to produce smarter, more flexible compilers — perhaps some day ones that generate code which adapts its representations of feature structures in response to the empirical distributions measured over several of the variables proposed here.

Section 2 enumerates the variables that we tested, and illustrates how some simple signatures change as a result of varying these dimensions of signature structure. Section 3 discusses the results of our experimental comparison of these variables, and Section 4 then focusses on the specific issue of fixed vs. variable frame allocation, and how these variables influence that choice. This can be seen as a case study in how these variables, and a static analysis of the signature more generally, can provide us with a deeper understanding of how grammars behave.

2 Dimensions of Signature Structure

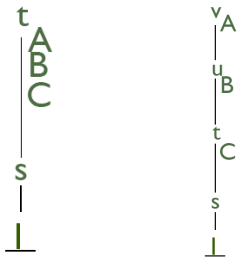
We measured the effects of varying each of the following dimensions of a signature's structure, both individually and together.

2.1 Arity Growth

Arity growth refers to how quickly a subtype chain confers additional appropriate features onto its types as a function of height. The signature in Figure 1(a) has faster arity growth than the signature in Figure 1(b). Both allocate the same number of appropriate features to their maximally specific type, but Figure 1(b) does so one at a time through a longer chain.

2.2 Chromatic density

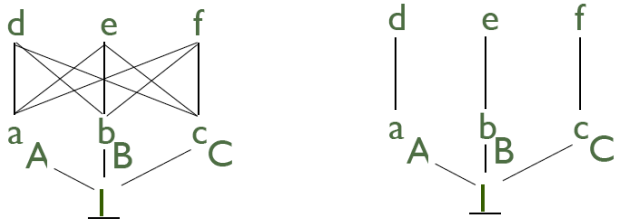
Signatures with a high chromatic density have more different pairs of features that are appropriate to a single type than signatures with low chromatic density. In Figure 2(a), for example, the pairs A and B, B and C, and A and C are all appropriate to some common type (three of them in each case, actually — d, e and f). In fact,



(a) (b)

Figure 1: An illustration of (a) fast arity growth vs. (b) slow arity growth.

all three together are appropriate to a common type. In Figure 2(b), on the other



(a) (b)

Figure 2: An illustration of (a) high chromatic density vs. (b) low chromatic density.

hand, there are still three features, but we will never find one of them in a feature structure where another is appropriate.

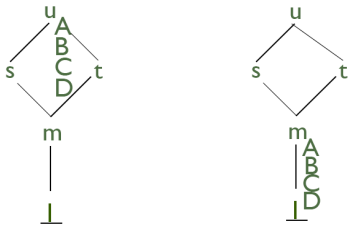
Signatures with low chromatic density require smaller frames in the fixed approach than signatures with high chromatic density (Penn, 1999).

2.3 Drag

Drag is related to chromatic density, but is also effected by how high within subtype chains the feature introducers are situated. In Figure 3(a), a fixed approach would need to assign as large of a frame to m as it does to u, in spite of the fact that m has no appropriate features of its own. m has a higher drag there than it has in Figure 3(b), because in Figure 3(b), it could use every slot that it is allocated by a fixed approach, in spite of the fact that its frame would be the same size.

2.4 Mesh

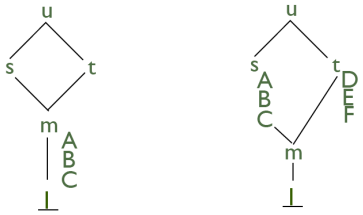
Mesh determines how many corresponding feature values must be (recursively) unified when two feature structures having a particular pair of types are unified.



(a) (b)

Figure 3: An illustration of (a) high drag vs. (b) low drag.

The pair s and t has a higher mesh in the signature of Figure 4(a) than it does in that of Figure 4(b), because, while they both have three appropriate features defined



(a) (b)

Figure 4: An illustration of (a) high mesh vs. (b) low mesh.

in both signatures, their sets of features are disjoint in Figure 4(b).

2.5 Static Typability

In a statically typable signature, the successful unification of two well-typed feature structures is always well-typed. In a non-statically typable signature, the results of successful unifications must be checked to ensure that they are. Figure 5 is a statically typable signature. Figure 6 is very close to Figure 5, but it is not statically typable, because the result of unifying feature structures of types s and t, even when successful, may not yield a feature structure that has an A value of type e. If successful, it will always yield a value at A that is consistent with e in the absence of inequations and extensional types, so it is often well-typable, even when not well-typed, but the addition of an extra maximally specific subtype of d to the signature could easily prevent even that. Non-statically typable signatures require more work to unify, in general.

Usually, we speak of an entire signature being statically typable or not, but we can easily generalize this to a degree of static typability by counting the number or percentage of type pairs for which unification would require this extra amount of type checking.

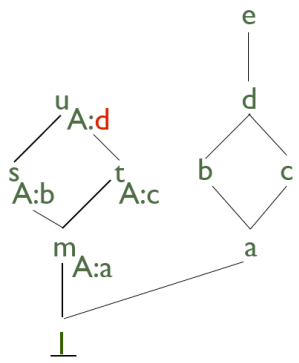


Figure 5: A statically typable signature.

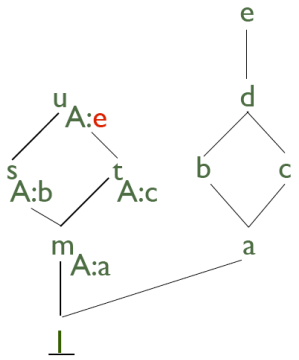


Figure 6: A non-statically typable signature.

2.6 Trailing

Trailing refers to the overhead of tracking a sequence of small changes to a data structure in memory (a *trail*) so that they can be undone in reverse order. Applications of backtracking search strategies often employ this. In the context of feature-structure-based all-paths parsing, backtracking can arise as a result of description-level disjunctions, subtype declarations in which a type has more than one immediate subtype (at least in some interpretations of subtyping), logic programs with predicate-level disjunctions or multiple clause definitions, or phrase structure rule systems in which the left-hand-side categories of two or more rules are unifiable.

Chart parsers almost by definition prefer the cost of structure copying to the cost of backtracking found in shift-reduce parsing, for example. Even within chart parsers, however, there are aspects of access to the parsing chart relative to which copying vs. trailing again trade off. This latter trade-off has been ignored for the most part, mainly because the re-discovery of dynamic programming within the

computational linguistics community happened to coincide with an infatuation for Prolog implementations of parsers, within which this kind of precise control over chart access was not available without a considerably greater amount of effort (Penn and Munteanu, 2003). Even with that effort — or without Prolog — the choice of copying vs. trailing is more crucial and more complicated to optimally resolve in the case of feature-structure-based parsing because of the size of the feature structures.

As for the other potential sources of backtracking, and therefore trailing, the trend within the HPSG community over the last thirteen or more years has been to mercilessly hunt them down and eliminate them. The English Resource Grammar at its inception deliberately ruled out the use of explicit disjunction operators, at the description or predicate level, for the sake of both efficiency and portability. The LKB, PET and later parsers adopted what was, at the time, ALE's very anomalous interpretation of subtyping and constraint resolution in order to back away from potentially very costly backtracking searches, curiously without the logic programming mechanisms that one needs in order to make this constraint resolution strategy sound and complete. We will return to this topic in Section 3.

We did not measure the cost of delaying (Penn, 2004).

3 Relative Cost

Given an abstract signature, such as one of the examples above, and a skeletal parsing control program, both of which can be modified to independently vary all of the parameters given in the last section, plus a constant underlying implementation of the unifier, we may first ask which parameter is inherently more costly than the other. Given a choice between making a grammar less chromatically dense or more statically typable, for example, which of these directions of development will result in a faster parser?

It is very difficult, and perhaps impossible, to answer this question in a way that generalizes over all grammars and all implementations. The parser implementation used in these experiments is described in great detail in Steinicke (2007). It is a reimplementation of the Warren Abstract Machine, modified to operate on typed terms that allow for subtype polymorphism, arity growth and non-static typability. It is written in C++ and was compiled with GNU C++ 4.1. All of the experimental runs described in this paper were run on an AMD athlon 64/3000 with 512 MB of RAM, and were iterated for 200,000 unifications per single time measurement reported. Approaching the implementation at this very low level allows us to rule out parochial properties of the memory managers used in higher-level programming languages, and focus on a single, fairly neutral implementation. The one very strong, although still common assumption made is that working (non-chart-edge) memory is allocated from a global stack that we maintain, in keeping with the architecture of the WAM. So we are doing something typical and reasonable, if not generalizable.

The base signature and parsing control are also described in Steinicke (2007). All of the experiments reported involve modifying aspects of these to vary the number of unifications, number of trail unwindings, size of the feature structures, etc. Again, this does not generalize over all grammars that linguists write, nor even look similar to a single grammar of a human language. There is also a serious concern with determining comparable units of measure along which each of these parameters varies. What we can do is spot asymptotic trends as these dimensions grow very large to formulate a neutral appraisal of their cost in the limit. The neutrality arises from our choice of implementation. The asymptotes allow us to generalize without committing to a single choice of units.

Asymptotically, then, the relative costs of these variables, in decreasing order are:

1. Static typability
2. Trailing
3. Chromatic density
4. Drag
5. Mesh
6. Arity growth

The relative ordering the same for fixed and variable frame allocation, although the disparity between them does change.

There are a few surprises here. Arity growth, arguably the most distinctive aspect of the logic of typed feature structures relative to other record logics, actually does not matter all that much. Also, (non-)static typability outranks even trailing in cost. This is interesting because non-statically typable signatures can also be unfolded so as to restore static typability, much in the same way that the English Resource Grammar's type system was unfolded to eliminate various sources of disjunction. Figure 7 shows the unfolding of Figure 6, for example. The English Resource Grammar did not do this, however, perhaps because there is no explicit operator in the description language for typed feature logic that can be held accountable for non-static typability. It arises from a conspiracy among several sources of appropriateness constraints.

Just so, what makes disjunctions dangerous is their ability to team up in networks to form NP-hard problems, not trailing specifically (although it is number two on our list). In fact, the presence of disjunctions does not even necessitate a backtracking search strategy.

4 Frame Allocation

Returning to the question of fixed vs. variable frame allocation, we can now consider this in the context of the variables that have been proposed. This is achieved

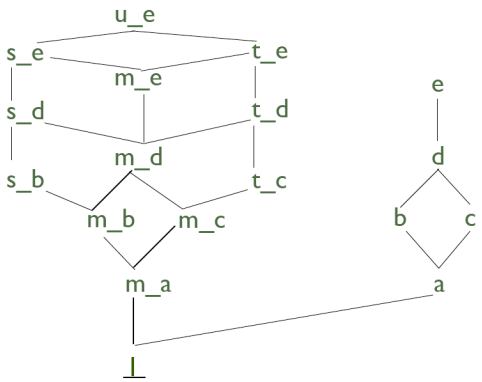


Figure 7: Eliminating non-static typability

by rerunning the above experiments, but now allowing the underlying implementation of the unifier to vary between the two allocation methods. In the case of the fixed method, graph colouring was used. The results are shown in Figure 8. Looking at both of the extremal cases, Figure 9 shows our experimental timings

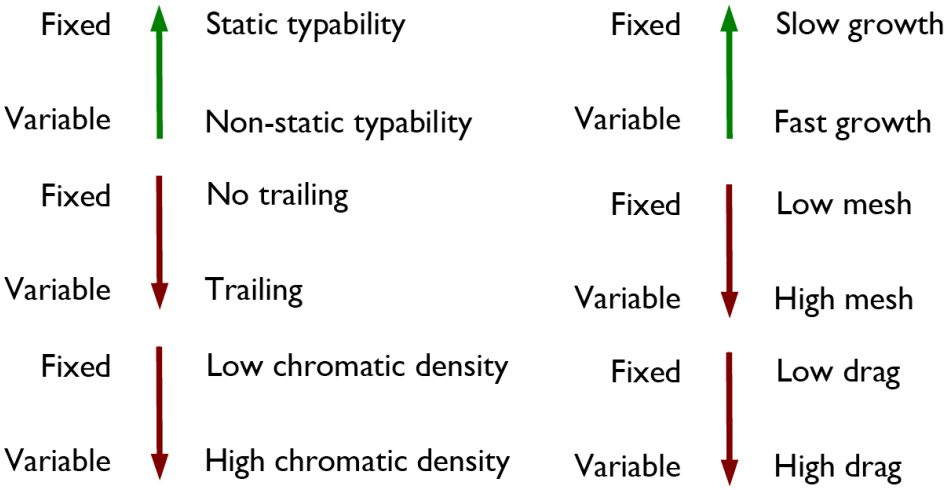


Figure 8: The influence of each variable upon the choice of fixed vs. variable frame allocation.

as the number of unused feature value slots increases for both the fixed and variable approaches when all of the variables are set to values that favour the variable approach. In this circumstance, the variable approach is clearly better. Figure 10 shows the same as the total number of features increases when all of the variables are set to values that favour the fixed approach. Here, the fixed approach is better,

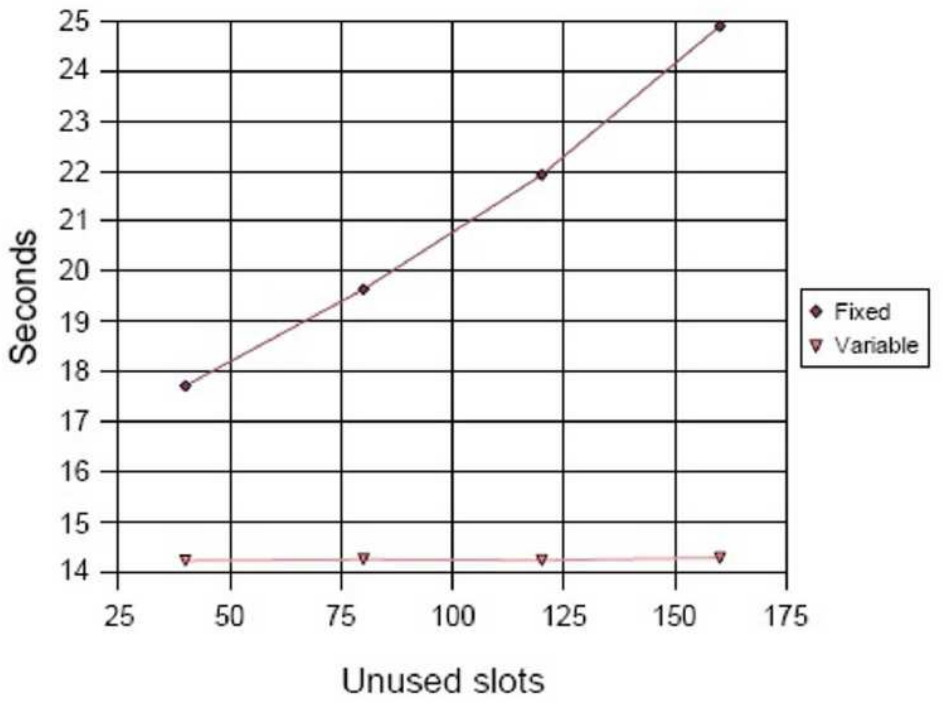


Figure 9: Experimental timings with every variable set to favour variable frame allocation.

but not by as wide of a margin. To illustrate the relative importance of trailing, Figure 11 shows the same measurement when all of the variables are set to the same values (favouring fixed), except that trailing on just one choice point is added. The presence of more trailing favours the variable approach. This one variable is enough to tip the balance. By the time the number of features exceeds 200, the fixed approach in this experiment was slower, in spite of the other variable settings. Figure 12 shows the same sort of inversion when the variables are all set to favour the fixed approach except that no pair in the unified types was statically typable. No other single variable setting results in an inversion on the size of features that we tested.

Turning to the English Resource Grammar again, Callmeier (2001) tells us that the variable approach is better with this grammar than a fixed approach with no graph colouring. Why might this be? The ERG is not at all statically typable (favouring the variable approach), has a very limited amount of trailing (fixed), a relatively high chromatic density across its different modules (variable), low drag (fixed), high mesh (variable), but fast arity growth (variable). We cannot simply count the properties that favour one or the other and decide on that basis — crucially, all of these are weighted by the empirical distribution of unification opera-

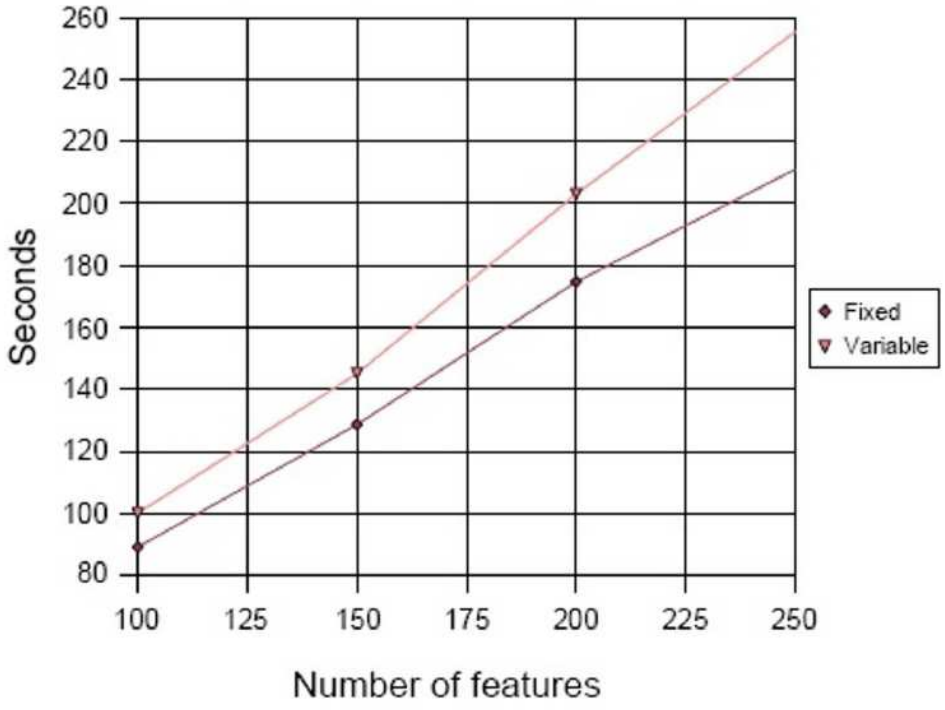


Figure 10: Experimental timings with every variable set to favour fixed frame allocation.

tions over pairs of types for the corpus that we use to evaluate the ERG. We did not calculate those weights. For what it is worth, however, the supervening importance of static typability and trailing provides no clear answer to this question, and in fact, the ERG is significantly faster with the fixed approach that ALE 4.0 introduced than with its earlier variable approach. ALE is written in Prolog, and there are doubtlessly many aspects of the Prolog compiler that favour the fixed strategy since Prolog terms themselves have fixed arities. Nevertheless, this does suggest that perhaps there is nothing about the ERG that strongly militates against either alternative, and that the choice in the case of ERG implementations ultimately hinges upon other design decisions.

5 Conclusion

We have identified a collection of the source-code level correlates of memory management costs evident in unifying typed feature structures. Since we can understand these primarily as structural properties pertaining to signatures (subtyping plus appropriateness conditions), they have the promise to guide grammar developers as well as system developers in building more efficient parsers. The structural

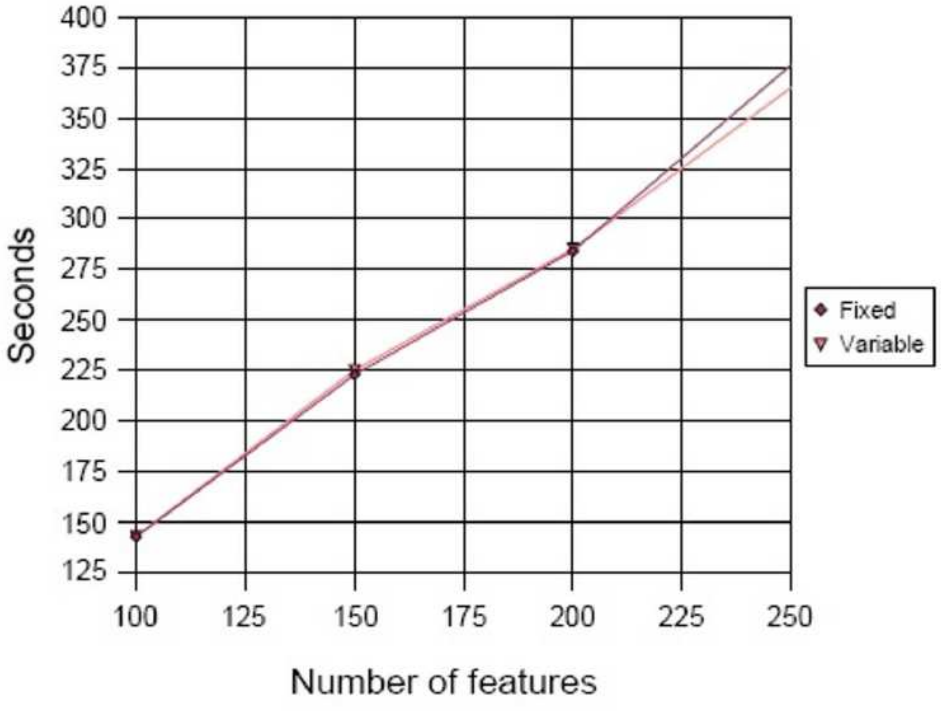


Figure 11: Experimental timings with every variable set to favour fixed frame allocation except trailing. One choice point is used.

properties complement our ability to embed grammars into existing systems and test their time and space efficiency on corpora. Static typability and trailing appear to be the most significant of these variables, in that by themselves they have the ability to override the settings of all of the other variables.

With respect to the specific issue of whether to use fixed-sized frames or variable-sized frames (that must then be resized), we can classify each of these variables according to its preference. With respect to the even more specific issue of which strategy to use with the ERG, we are unable to make a definite conclusion. Very clearly, the next step in demonstrating the value of our proposed sort of analysis would be to collect distributional data from the unifier input during parsing with a large grammar like the ERG, in order to show that our static analysis combined with these empirical data have the ability to definitively predict various resource consumption aspects of the parsing task.

References

Callmeier, U. 2001. *Efficient Parsing with Large-Scale Unification Grammars*. Masters Thesis, Universitaet des Saarlandes.

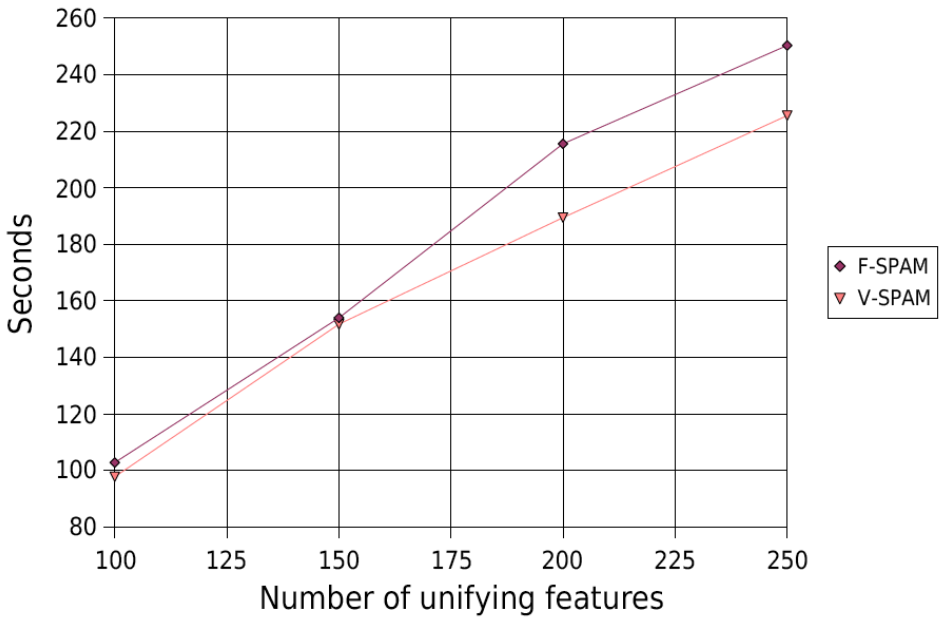


Figure 12: Experimental timings with every variable set to favour fixed frame allocation except static typability. No pair of unified types was statically typable.

Carpenter, B. and Penn, G. 1996. Compiling Typed Attribute-Value Logic Grammars. In H. Bunt and M. Tomita (eds.), *Recent Advances in Parsing Technologies*, pages 145–168, Kluwer.

Copestake, A. and Flickinger, D. 2000. An open-source grammar development environment and broad-coverage English grammar using HPSG. In *Proceedings of the Second conference on Language Resources and Evaluation (LREC-2000)*.

Goetz, T. 1993. *A Normal Form for Typed Feature Structures*. Masters Thesis, Universität Tübingen.

Makino, T., Torisawa, K. and Tsuji, J. 1998. LiLFeS — Practical Unification-Based Programming System for Typed Feature Structures. In *Proceedings of the 36th Annual Meeting of the Association for Computational Linguistics and the 17th International Conference on Computational Linguistics (COLING/ACL-98)*, volume 2, pages 807–811.

Penn, G. 1993. *A Utility for Typed Feature Structure-based Grammatical Theories*. Masters Thesis, Carnegie Mellon University.

Penn, G. 1999. An Optimized Prolog Encoding of Typed Feature Structures. In *Proceedings of the 16th International Conference on Logic Programming (ICLP-99)*, pages 124–138.

Penn, G. 2000. *The Algebraic Structure of Attributed Type Signatures*. Ph. D.thesis, Carnegie Mellon University.

Penn, G. 2004. Balancing Clarity and Efficiency in Typed Feature Logic through Delaying. In *Proceedings of the 42nd Annual Meeting of the Association for Computational Linguistics (ACL-04)*, pages 240–247.

Penn, Gerald and Munteanu, Cosmin. 2003. A Tabulation-Based Parsing Method that Reduces Copying. In *Proceedings of the 41st Annual Meeting of the Association for Computational Linguistics*, pages 200–207.

Steinicke, K. 2007. *Memory Management for Logic Programming with Typed Feature Structures*. Masters Thesis, Universität Tübingen.