Memory Management for Unification-based Processing of Typed Feature Structures

Karen Steinicke
Dept. of Computational Linguistics
University of Tübingen
72074 Tübingen, Germany
ks@sfs.uni-tuebingen.de

Gerald Penn
Dept. of Computer Science
University of Toronto
Toronto, M5S 3G4, Canada
gpenn@cs.toronto.edu

1 Introduction

A major design decision in the development of HPSG parsers is the internal representation of typed feature structures. Two approaches have been advanced relative to heap architectures that support unification: one by Carpenter and Qu Carpenter and Qu (1995), variations of which were adopted by both LiLFeS (Makino et al., 1998) and PET (Callmeier, 2000), in which feature structure representations grow as the type becomes more specific and more features become appropriate; the other by ALE (Penn, 1999b), in which space is reserved in advance on the heap for future appropriate features, and the minimal amount of space necessary is calculated at compile-time. No one except Callmeier (2000) conducted any experiments to corroborate their choice of representation, and Callmeier (2000) only briefly summarizes his findings, which were based on experimentation with one grammar (the English Resource Grammar, ERG).

What is at stake is not a time-space tradeoff. Either approach consumes more or less of both time and space than the other as a function of the type hierarchy's shape, feature introductions, and the empirical frequency distribution of feature structure instances over types. While empirical frequency distributions are only forthcoming from real large-scale grammars, of which there are precious few in HPSG that have not been directly or indirectly adapted from the ERG, a more controlled comparison that varies only some algebraic properties of abstract type hierarchies and feature introduction is also illuminating, since it provides prior guidance on how the type hierarchies of large-scale grammars should look for the sake of efficiency. Naturally, there are linguistic grounds for preferring one type system over another as well, but there can be cases in which the data, even when combined with our theoretical viewpoint, do under-determine the combination of features and subtyping used in our analyses. In these cases, efficiency is the natural criterion to base our decision upon.

This paper presents such a comparison, describing the platform on which both memory management strategies were implemented, called SPAM, as well as the results obtained from this experiment. SPAMis a modified reimplementation of the Warren Abstract Machine (WAM, Warren, 1983) that takes into account the inclusionally polymorphic aspects of processing typed feature structures. It has a very rigid Prolog-like description language, called Signature Prolog, although it lacks much of ALE's functionality. As a result, at the grammar specification level, it does not look like TDL, ALE or any other grammar design system, nor was it designed as an alternative to these. It simply provides us with sufficiently low-level access to memory management to conduct this experiment. A commercial implementation of Lisp or Prolog would not. PET imposes additional restrictions regarding static typability that limit the range of type systems that we can compare.

The conclusion apparent from our experiment, in brief, is that the choice of memory management strategy depends principally on: (1) the degree of static typability (Carpenter, 1992), [1] and (2) the amount of trailing required by the parsing algorithm or program. [2] ALE's "fixed" method performs better when the degree in (1) is high and the degree in (2) is low. Carpenter and Qu's "variable" approach performs better when the degree in (1) is low and the degree in (2) is high. The constant terms arising from our own implementation and machine architecture preclude our setting statistically significant thresholds beyond which one approach overtakes the other, but these trends are robust in relative terms, i.e., as a type system becomes less statically typable and processing requires more trailing, Carpenter and Qu's method becomes more preferable.

[1] The definition of static typability in Carpenter (1992) does not come with a notion of degree or gradation, but by this we simply mean to measure the percentage of type unifications that do not require dynamic type inference. The classical sense of static typability is obtained when this percentage converges to 100% with increasingly large samples from a uniform distribution of unifications over pairs of types in the hierarchy.

[2] Trailing records the changes made to feature structures on the heap so that they can be restored during backtracking. Even if the description language lacks an explicit disjunction operator, as in PET or the LKB, all-paths parsing algorithms implicitly backtrack by considering alternative parses of the same substrings. Not all backtracks involve trailing, however.

2 Architecture

SPAM adopts the general scheme of compiling programs into a sequence of *abstract machine instructions* from the WAM. Our modifications to the WAM basically affect the way arguments to functors are processed: functors are interpreted as types and their arguments as feature values. Whereas this necessitates creating an enirely new unification mechanism, program resolution and parsing are by and large identical to what a Prolog compiler would use.

The most signficant syntactic modification to the input language is that the user must place a type system specification in front of the parser/program. A *type system*, or *signature*, comprises a type hierarchy, a finite set of features and an appropriateness specification (see Carpenter, 1992). The type hierarchy is compiled into a finite bounded complete partial order, as is standard in many HPSG processing systems. Apart from this, Signature Prolog (programs and queries) syntactically look very much like Prolog. SPAM's memory layout includes all the memory areas employed by a classical WAM, the most important of these being the *heap*, an addressable data structure consisting of tagged cells, where unification of feature structures takes place.

This test platform can accommodate two approaches as described above: a *fixed* approach and a *variable* approach. In the following, the machine and all its components will be prefixed with *V*- or *F*- according to the approach. Each approach includes a respective machine (*F*-SPAM vs. *V*-SPAM), its respective machine language, and the respective compiler that compiles Signature Prolog into the respective machine language.

In SPAM, a node (substructure) of a feature structure together with the edges (features, if any) that depart from that node is represented by a *frame* on the heap. The nodes pointed to by these edges are also included in the frame if they do not themselves have edges departing from them. The representation of a feature structure on the heap is a collection of such frames.

A frame consists of a *root cell* and a contiguous block of cells adjacent to the root cell, that are also referred to as *slots*. The root cell contains type information and serves as a location in memory to which other typed feature structures can point — this location is the unique representative of this feature structure's identity. In *V*-SPAM, the number of allocated slots is identical to the arity of the node residing in it, i.e., the number of features appropriate to its type. In *F*-SPAM, the number of allocated slots might be greater than in the former, i.e. not every slot of the frame might be occupied. The slots that are not occupied by a feature structure remain unused until they are needed later, if ever.

Feature structures are *polymorphic* data structures in that the type of a feature structure as well as the number of its features, and also the appropriate types of those features' values might change during unification. Thus, in *V*-SPAM, if a node's type is promoted to a more specific type with greater arity, the corresponding to it "outgrows" its frame, since it can acquire one or more new features. This situation will be referred to as a *growth-situation*. It is a source of both time and space complexity. In a growth-situation, all that can be done in *V*-SPAM is to assign new space (viz. a larger frame) to the resulting larger frame, and to relate the larger frame to the original frame by pointers. To determine how the features of the original frame(s) and the larger frame match up, *V*-SPAM needs to refer to a *types* × *types* table (viz. the *V*-`FRAME_UA`below).

The fixed approach is characterized by the frame size being fixed – more precisely, fixed per module (see below). Furthermore, every feature has a fixed position within every frame. Thus, in many situations, no run-time table lookup is necessary (see section 3.1) to find out how features of unificands match up. At the end of the computation it might be the case that not every slot allocated to a frame is occupied. However, the size of a frame remains constant, and no redirection from any new frame to an original frame is necessary. To ease the problem of potentially very large frame sizes (depending on the number of features to be accomodated), *F*-SPAM adopts both the *modularization* and *graph coloring* optimizations discussed in Penn (1999b). A situation in which more unused than used slots are carried around will be referred to as a *drag-situation*. This is the alternative source of time and space complexity for this approach. Neither of the approaches constitutes a perfect solution, but, depending on the application and the signature, one approach is more suitable than the other.

2.1 Static areas

All areas of the WAM's memory architecture are adopted, some with minor modifications. SPAM also has additional data areas that matter for the experiment in section 4. These are the *signature area* and the *assert area*, for dynamic assertion of fact clauses such as chart edges. SPAM's memory areas fall into *static* and *dynamic* areas. The static areas are the *signature area* and the *code area*. At compile-time the signature is compiled into the signature area, and the program/parser is compiled into abstract machine instructions that are stored in the *code area*.

In the signature area, all kinds of information about types as well as of unification of types and of feature structures is stored. The relevant components of the signature area we'll need in section 4 are the *`TYPE_UA`* and the *`FRAME_UA`*. The `TYPE_UA` (type unification area) is a *types* × *types* table (roughly the

same table for both F-SPAM and V-SPAM) storing for each pair of types their least upper bound as well as the relation between them: proper join, or one of two subtype relationships.

The FRAME_UA (frame unification area) is a table used for unification involving at least one complex feature structure. The V-FRAME_UA is a *types* × *types* table containing for each pair of unifying types t_1 and t_2 a small table *fc* (feature unification code), which describes, for each feature in the unification result, its resulting value as a function of the slots in the arguments of types t_1 and t_2. The encoded instructions consist of an *origin*, which is either t1, t2, both or neither, referred to as "orig" in Figure 2, and the appropriate type of the resulting value. The F-FRAME_UA contains for each type t a *slot restriction* (*sr*) determining the position and the appropriate type of every feature in a frame of type t.

3 Run-time System

The dynamic areas that are relevant for section 4 are the heap, the *stack*, the *trail* and the assert area. They are involved at run-time only. This is when space for them is allocated.

The heap consists of tagged data cells with the tags: STR (the tag of the root cell of a feature structure), VAR (the tag of a type or an unbound variable) and PTR (the tag of a cell that points to a cell tagged VAR or STR). In F-SPAM, unused slots are tagged EMPTY. H is a *global register* containing the next available address on the heap.

A failure of unification leads to backtracking in case there are alternatives to consider in program resolution. The state of computation at a program procedure call which offers alternatives is referred to as a *choice point*. The information wherefrom such an original state can be restored is pushed onto a stack (more precisely, onto an *OR-stack*, see Aït-Kaci (1999)).

The trail's purpose is — in case a choice point was pushed – to save a heap cell's contents before it is changed. The contents of both VAR and STR cells are potentially *bound* and *trailed*. A global register *HB* is set to contain the value of H at the time of the latest choice point. Only bindings of cells whose addresses are less than HB need to be recorded in the trail. Upon reconsidering a choice, the trail needs to be *unwound* to reconstruct the original state.

3.1 Unification

The notion of *adding* a unificand to another will be used below to indicate unification, with a certain direction or bias in the order of its arguments. One unificand always resides on the heap and will be referred to as the *add_target*; the other unificand may reside in the code area, on the heap, or in the signature area, and will be referred to as the *addend*. We will use the term, "unification" whenever we want to abstract away from this directionality. The tests presented in section 4 focus on the case where both the addend and the add_target reside on the heap. This situation will also be referred to as *frame unification*.

The signature depicted in Figure 1 is a *non-statically typable signature*, due to the fact that the value restriction for F_1 on type c is d_4 (and not d_3) which is strictly more specific than would be required to maintain right monotonicity (see Carpenter, 1992), given the value restrictions that F_1 has at supertypes a and b. To find the value restriction for F_1 on type c, *dynamic (or run-time) type inferencing* is needed. To obtain a totally well-typed result when unifying feature structures of types a and b, the feature F_4 (which is introduced by c) needs to be filled in (see Carpenter, 1992). Whereas a V-FRAME_UA-lookup is

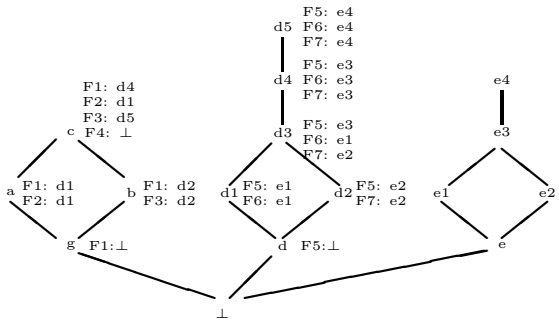

Figure 1: **A non-statically typable signature**

always needed to unify complex feature structures of differing types, an F-FRAME_UA-lookup is needed only in the case of dynamic type inference and when filling in missing features becomes necessary.

When unifying (see heap contents in Figure 3) two feature structures of root types d_1 and d_2, V-SPAM looks up the fc (see Figure 2) for the pair $\langle d_1, d_2 \rangle$ which guides the unification process. Here, the type relation t3_GROWS indicates that d_1 and d_2 unify to a new type with greater arity, viz., a growth situation. Then the origins follow: UNIFY indicates that both unificands' values (of feature F_5) must be unified; COPY_t1 indicates, that the unification result must contain a pointer to F_6's value (only d_1 bears F_6); similarly COPY_t2 indicates that the unification result must contain a pointer to F_7's value.

Since neither filling in missing features nor dynamic type inference is needed, while stepping through the unficands' frames on the heap, F-SPAM (see heap content in Figure 4) needs only to compare the corresponding slots of the two frames, where the *slot patterns* of the unificands can be seen to correspond to the origins in Figure 2; i.e. NON-EMPTY – NON-EMPTY (at addresses 3002 and 3006) corresponds to UNIFY, NON-EMPTY – EMPTY (at 3003 and 3007)

to COPY_t1 and EMPTY – NON-EMPTY (at 3004 and 3008) to COPY_t2. In both Figure 3 and 4, the unification result resides in the respective frames of type d_3.

	rel/origin	lub/restr
0	t3_GROWS	c
1	UNIFY	e3
2	COPY_t1	e1
3	COPY_t2	e2

Figure 2: **fc for pair of types <d1,d2>**

Figure 5 depicts the fc for the pair $\langle a, b\rangle$ in V-SPAM and the sr for type c in F-SPAM, where a and b involve both dynamic type inference and the filling of feature F_4. The suffix "_D" indicates the necessity for dynamic type inference; CREATE indicates that a feature must be filled in. When unifying feature structures of types a and b, V-SPAM makes one V-`FRAME_UA`-lookup, whereas F-SPAM needs an F-`TYPE_UA`-lookup to determine the resulting type and type relation and then an F-`FRAME_UA`-lookup to obtain the sr.

An fc needs more space than an sr but is much more precise: for each feature, it specifies whether it is introduced by the unification result and whether dynamic type inference is needed, along with the value restriction of the respective feature appropriate to the unification result. Each sr entry says neither which features' values need dynamic type inference nor which features are introduced by the unification result — only whether dynamic inference is necessary somewhere.

4 Controlled Experiment

The experiments were conducted in such a way that a *minimal application* accessing *one small fragment of the signature* was repeated many times, as we are particularly interested in testing growth- and drag-situations. Therefore a failure-driven loop written in Signature Prolog serves as the testing environment, executing the unification of two feature structures a

⋮	⋮	⋮
3001	STR	d1
3002	VAR	e1
3003	VAR	e1
3004	STR	d2
3005	VAR	e2
3006	VAR	e2
⋮	⋮	⋮

3001	PTR	3007
3002	PTR	3008
3003	VAR	e1
3004	PTR	3007
3005	PTR	3008
3006	VAR	e2
3007	STR	d3
3008	VAR	e3
3009	PTR	3003
3010	PTR	3006

Figure 3: **Heap before (to the left) and after (to the right) unification in *V*-SPAM**

3001	STR	d1
3002	VAR	e1
3003	VAR	e1
3004	EMPTY	
3005	STR	d2
3006	VAR	e2
3007	EMPTY	
3008	VAR	e2

3001	PTR	3005
3002	PTR	3006
3003	PTR	3007
3004	EMPTY	
3005	STR	d3
3006	VAR	e3
3007	VAR	e1
3008	VAR	e2

Figure 4: **Heap before (to the left) and after (to the right) unification in *F*-SPAM**

	rel/origin	lub/restr
0	t3_GROWS	c
1	UNIFY_D	d4
2	COPY_t1	d1
3	COPY_t2_D	d5
4	CREATE	bot

	pos	restr
0	1	d4
1	2	d1
2	3	d5
4	4	bot

Figure 5: **fc for pair of types $\langle a, b\rangle$ and sr for type c**

large number of times while guaranteeing a bound on heap memory consumption. The initial input is remembered by asserting it to the assert area, to enable repetition of the same unification whilst avoiding continuation with the instantiated query after the first execution of the loop body. This looks as shown in Figure 6.

```
add(List_1):-
  clause(edge(1, 2, List_2)),
  rep_add__2(List_1, List_2),
  1=2.          %fail
add(_).

rep_add__2([],_).
rep_add__2(_,[]).
rep_add__2([T1|Rest1],[T2|Rest2]):-
  T1 = T2,      %unification
  assert(edge(2, 3, [T1])),
  rep_add__2(Rest1,Rest2).
```

Figure 6: Loop body

To put frame unification to the test, the query that the user is supposed to enter should be something like the one shown in Figure 7. where "=" in `add/1` invokes adding the addend (the argument of the second list) to the add_target (the argument of the first list).

`add/1` sets an (artificial) choice point. F-SPAM runs the risk of trailing, since cells are more likely to reside underneath `HB` than in V-SPAM. Tests were conducted both in the presence of the second `add/1` clause, and in its absence.

The tests were conducted with many features and uniform feature values. The has the effect of magnifying the number of computations so that the difference can be precisely measured, and at the same time controlling the experiment for other variables that could

```
?-n_times(20000, [dt1(at1,at1,at1,at1,at1)],
                 [dt2(at2,at2,at2,at2,at2)]).
```

Figure 7: Example query

kind of origin used	without 2nd add	with 2nd add
UNIFY	**pro-F (90.3%)**	pro-V (96.67%)
COPY_t2	**pro-F (80.95%)**	same
COPY_t1	**pro-F (94%)**	same
UNIFY_D	pro-V (94%)	**pro-V (86.02%)**
COPY_t1_D	same	**pro-V (92%)**
COPY_t2_D	pro-V (94%)	**pro-V (90%)**
CREATE	pro-V (98%)	**pro-V (76%)**

Figure 8: Basic summary of results

confound the single effect we are looking for. The relative performance of the two approaches depends both on the application and the signature. Both the attributes of the signature and the attributes of the application varied in the tests. The application-inherent attributes that should be mentioned here are: presence vs. absence of the choice point, of growth situations and of drag situations. The relevant signature-inherent attributes were distribution of origins (see Figure 8) and degree of statically typability.

V-SPAM's fc's can be seen as the interface between the application and the signature, in that the fc is determined by the signature fragment that is used in the minimal application. There are no fc's in *F*-SPAM, but since the origins correspond to slot patterns in *F*-SPAM, we can describe the experiment in terms of fc's in both approaches.

Figure 8 breaks down the combinations of fc and choice point presence, i.e., presence of the second add clause, during frame unification. The first column shows the main kind of origin used in the respective fc to characterize the fragment of the signature area used by the minimal application. pro-F means that the fixed approach scored better, and pro-V, that the variable approach scored better for the respective distribution of origins. Percentages are of the faster time over the slower. Boldfacing means that the result is statistically significant. The longest of these (UNIFY_D with choice points in *F*-SPAM) took 157.56 seconds. The shortest (COPY_t2 without choice points in *F*-SPAM) took 31.2 seconds. Again, "_D" indicates dynamic type inference.

What the origins mean in the variable approach, is detailed in section 3.1. In *F*-SPAM, COPY_t2 means that the respective feature value of the add_target is left untouched; COPY_t1 means that the respective feature value resides in the addend and thus must be related to the add_target via a PTR cell placed in the add_target; and UNIFY is interpreted the same in both approaches. If very many features combine but their values are not changed, the fixed approach performs better.

As discussed in 3.1, dynamic type inference and filling features (CREATE and suffix "_D") slow down the fixed approach.

After having passed a choice point, more changes to structure under HB slow the fixed approach down. This effect is certainly a result of the data structure used for trailing. Upon backtracking, the respective slot must be reset to EMPTY, which slows down the fixed approach. This is related to drag situations. If the empty slots are at the bottom of the frame, in the absence of the choice point, they do not have much negative influence on performance, but the optimal graph coloring does not always allow for this possibility.

5 Conclusion

Whether variable or fixed frame encodings of features in typed feature structures are more efficient depends on whether (1) the type signature in question is statically typable and (2) the logic program in question requires trailing. On a sliding scale from "(1) but not (2)" to "(2) but not (1)" (the slide coming from the presence of more or less trailing and the amount of dynamic type inference required, if any), our preference shifts from fixed frames to variable frames, respectively.

References

Aït-Kaci, Hassan 1999. Warren's Abstract Machine — A Tutorial Reconstruction. *Note:* reprinted from MIT Press version.

Callmeier, U. 2000. A platform for experimentation with efficient hpsg processing techniques. *Natural Language Engineering*, 6.1:99–108. *Note:* Special Issue on Efficient Processing with HPSG.

Carpenter, Bob 1992. *The logic of typed feature structures*. Cambridge: Cambridge Univ. Press.

Carpenter, B. and Qu, Y. 1995. An abstract machine architecture for typed attribute-value grammars. In *Proceedings of the 4th International Workshop on Parsing Technology*.

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)*, 807–811.

Penn, Gerald 1999a. An optimised prolog encoding of typed feature structures. Arbeitspapiere des SFB 340 138, Universität Tübingen, Germany.

Penn, Gerald 1999b. An optimized prolog encoding of typed feature structures. In D. DeSchreye (ed), *Logic Programming: Proceedings of the 1999 International Conference on Logic Programming*, 124–138, Cambridge, MA: MIT Press.

Warren, David H. D. 1983. An abstract prolog instruction set. Technical Note 309, SRI International, Menlo Park, CA.