Arbeitspapiere des SFB 340
Bericht Nr. 138, January 1999

AN OPTIMIZED PROLOG ENCODING OF TYPED FEATURE STRUCTURES

GERALD PENN

Abstract. A new Prolog-term-encoding of typed feature structures, a data structure commonly used for natural language processing, is presented that, in many significant cases, allows for faster unification in Prolog implementations of feature-structure-based natural language processing systems, e.g., ALE (Carpenter and Penn 1998). It remains general enough, however, to handle the full range of signatures characterized in Carpenter 1992, a common reference on typed feature logics for computational linguistics, and HPSG (Pollard and Sag 1994), a common linguistic theory based on typed feature structures. A logical and empirical comparison is drawn with several alternative encodings.

Contents

1. Introduction: Typed Feature Structures

Typed feature structures (TFSs, Carpenter 1992) are like the record structures found in many programming languages, or the "frames" that are used in artificial intelligence. They are collections of pairs of attributes and values along with a semantic type, that characterize or classify an object for the purposes of some computation. Feature structures of many varieties have enjoyed a wide use in linguistics over the past 30 years, and at every level of linguistic reasoning (syntax, phonology, etc.). The main

ISSN 0947-6954/99

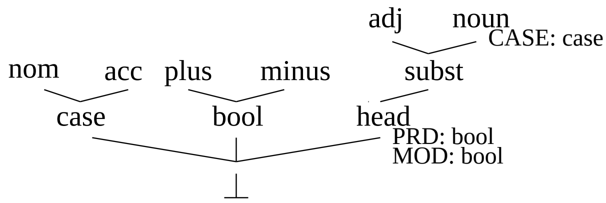

FIGURE 1. A sample type signature.

reasons for their popularity are: (1) their support for named reference of attributes, in contrast to the exclusively positional reference provided by Prolog terms and (2) the inclusional polymorphism provided by their type system. The former allows for terse descriptions of very large structures; and the latter provides an additional means of classifying the kinds of (linguistic) knowledge represented by those structures.

When programming with TFSs, one must define the types and features available to structures ahead of time by specifying a *type signature*, consisting of a partial order of types (*type hierarchy*) and a set of *appropriateness conditions* that constrains features to take values on all and only feature structures of particular "appropriate" types, and only values that are themselves of a certain type (*value restrictions*). A sample type signature is shown in Figure 1.[1] All feature structures of type *head* must have values for all and only the features PREDICATIVE (PRD) and MODIFIER (MOD), with values of at least type *bool(ean)*. Subtypes *inherit* their supertypes' appropriate features, so that *noun* also has PRD and MOD values, in addition to a value for CASE. The feature logic of Carpenter 1992 is, in fact, distinguished from other feature logics by its very strict interpretation of appropriateness conditions. This strict interpretation allows for more efficient processing with TFSs because, for every type, the exact set of these appropriate features that can and must occur with feature structures of that type can be computed at compile-time. TFSs can, in fact, be viewed as a conservative extension of Prolog terms in which, for any single type, the feature structures of that type have a fixed arity; but the arity of a feature structure can be augmented as its type promotes to a more specific subtype. Such is the case for a *subst*-typed FS that

[1]In keeping with the choice of notation in Carpenter 1992, $\perp$ will stand for the most general type, and supertypes are depicted below their subtypes. In that setting, the unification of two types corresponds to their join, where it exists, and is undefined otherwise.

promotes to *noun*, thereby acquiring a value for CASE, restricted to at least *case*.

Appropriateness is related, although not identical to the use of arity constraints in the LIFE/Half_Life and Oz systems (Aït-Kaći 1993; Meyer, Dumant, and Van Roy 1994; Van Roy, Mehl, and Scheidhauer 1996). Crucially, feature-based linguistic theories, notably Head-driven Phrase Structure Grammar (HPSG, Pollard and Sag 1994), either explicitly or implicitly observe the constrained arity property embodied in appropriateness conditions, whereas their restricted use of features cannot be encoded by LIFE/Oz-style arity constraints without substantial rewriting of the type signatures involved.

Appropriateness also allows us to encode typed feature structures as Prolog terms efficiently. Arity augmentation and subtyping are the two major sources of complexity when encoding TFSs as Prolog terms. This paper presents a collection of methods to address both sources by reducing the former to a graph coloring problem, and by presenting for the latter what is, to the author's knowledge, the first algorithm for finding an optimal flat first-order-term encoding of any finite meet semi-lattice of types, and an approximate solution that can be derived in cubic time in the number of types from a transitively closed adjacency representation of the semi-lattice. An empirical comparison of these to previous work as well as other encodings that are available using the extra functionality provided by the `library(atts)` library of SICStus Prolog is also presented.

The value of Prolog encoding stems from the value of using Prolog itself. Most systems based on TFSs rely on a few standard means of search in order to solve problems in natural language processing — some subset of Prolog-like SLD resolution, parsing and content-driven generation. All of these have been extensively researched within Prolog; and, in the case of SLD resolution, current commercial implementations of Prolog have benefited from a 15 year history of optimizations to WAM compilation technology. There have been a few abstract machines proposed directly for TFSs, e.g., Carpenter and Qu 1995, implemented as described in Makino, Torisawa, and Tsuji 1998, and Wintner 1997; but these inevitably rely on a recapitulation of that history for their own optimization. With the arguable exception of the unification operation itself, any additional innovations made in the course of their development are probably better applied to WAM optimization itself, given the nearly identical requirements of the two communities and research programs. The approach to arity augmentation presented here, however, would be equally useful to custom-abstract-machine-based approaches; and the present approach to subtyping essentially brings Prolog-based implementations in line with the bit-vector encodings for type unification used in many abstract machines stemming from the influential paper (Aït-Kaći, Boyer, Lincoln, and Nasr 1989) on that subject.

2. Subtyping

In this section, we consider type signatures without features — just type hierarchies. Carpenter (1992) requires type hierarchies to be finite meet semi-lattices. When two TFSs are unified, their types must unify. In a finite meet semi-lattice, every consistent pair of types, t_1, t_2 has a unique join, $t_1 \sqcup t_2$, which corresponds to type unification.

Here, we take a "Prolog encoding" to mean one in which the only operation necessary for FS unification at run-time is Prolog unification of the corresponding terms. That excludes other "Prolog representations," such as the representation of feature structures in ALE, a logic programming language based on TFSs, which requires a dereferencing operation and table look-up at run-time (Carpenter and Penn 1996), as well as the representation given in Gerdemann 1995, which because of its slightly different interpretation of appropriateness conditions, requires the maintenance of extra constraints on the side in the worst case. The actual construction of any Prolog encoding can be performed at compile-time.

2.1. Previous Work: Tree Encodings.

The first work to consider Prolog encodings of arbitrary meet semi-lattices was presented in Mellish 1991, although no general encoding algorithm was presented. Previous work, dating back to Dahl 1977, concerned a restricted subset of semi-lattices that admit a Prolog *tree encoding*, i.e., an encoding by terms in which no variable is used more than once. These and other logical-term-encoding approaches are systematically presented in Fall 1996.

Mellish 1991 was also the first to characterize the general encoding problem formally: the search for a function from a type hierarchy to a set (of Prolog terms) that is:

1. injective, i.e., no term encodes more than one type,
2. homomorphic with respect to unification, i.e., unification in the encoding corresponds to unification in the type hierarchy, and
3. zero preserving, i.e., failure of unification in the type hierarchy corresponds to failure of unification in the encoding.

What is now regarded as the most efficient tree-encoding algorithm was first presented in Mellish 1988 as a general encoding algorithm for systemic

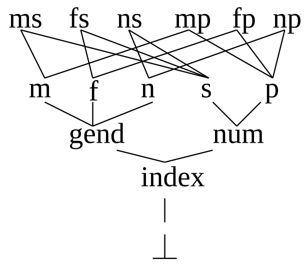

FIGURE 2. A type hierarchy with full-product multiple inheritance.

networks. All systemic networks are tree-encodable because the "multiple inheritance" that they effectively provide is always a full product of two or more sub-networks. The ProFIT system (Erbach 1994; Erbach 1995; Erbach 1996), another TFS-based system, first adapted this encoding to a subset of type hierarchies expressed in a more traditional ISA-link form. The ALEP system (Simpkins and Groenendijk 1994) used essentially the same encoding for its TFS-like records.

Mellish's encoding (Mellish 1988) exploits the restricted inheritance provided by systemic networks to represent a type with a term that, using subterms, represents the path(s) taken to reach that type from the root. In Figure 1, the representation of *head* would be `head(_)`; that of *subst*, `head(subst(_))`; and that of *noun*, `head(subst(noun))`. The `noun` subterm does not require a variable argument because *noun* is maximally specific.[2]

To represent multiple inheritance, the tree-based encoding uses multiple argument positions to represent the different paths that lead to a single type. In Figure 2, any pair of *gend* and *num* subtypes can intersect. So *index*'s term contains two argument positions, one for *gend* and one for *num*. The representations of *gend*, *s* and *ms* are `index(gend(_),_)`, `index(_,num(s))` and `index(gend(m),num(s))`, respectively.

Tree-based encoding does not work for arbitrary finite meet semi-lattices, as proven in Mellish 1991; Mellish 1992. Figure 3, for example, represents a simple classification of vowels, taken from Hudson 1981 (and cited in Mellish 1991). There are vowels that have any pair of the three properties, *lax*, *central* and *centering*; but there are no vowels that have all three at once. This has no tree encoding because separate argument

[2]In practice, an extra variable argument is necessary to distinguish feature values that are variants from feature values that are extensionally identical, because the logic of typed feature structures is intensional. These are systematically ignored here.

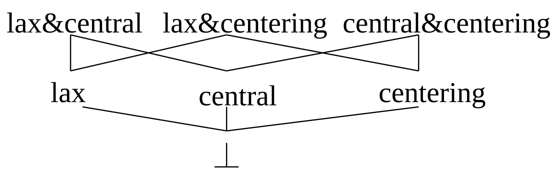

FIGURE 3. A type hierarchy with no tree encoding.

positions for *lax*, *central* and *centering* would entail the consistency of *lax* with *central&centering*, for example.

ProFIT dealt with this limitation simply by restricting its type hierarchies to exclude all counter-examples. Specifically, it allowed for two kinds of declarations:

$$Super > [Sub_1, Sub_2, \ldots, Sub_n].$$
$$Super > [Sub_{1,1}, \ldots, Sub_{1,n_1}] * \ldots * [Sub_{m,1}, \ldots, Sub_{m,n_m}].$$

The first implicitly declares $Sub_1, \ldots, Sub_n$ to be mutually exclusive, i.e., no multiple inheritance. The second declares *multi-dimensional inheritance*, essentially restricting multiple inheritance to the full products of types in different dimensions attainable in systemic networks. All possible combinations of dimensions must be possible in such hierarchies, so again, Figure 3 cannot occur. We could declare the subhierarchy rooted at *index* in Figure 2 as:

$$index > [gend] * [num].$$
$$gend > [m, f, n].$$
$$num > [s, p].$$

The maximally specific types, *ms,fs* etc., are implicitly encoded by the intersection of their *gend* and *num* subtypes.

2.2. **Flat-Term Encoding.** The present work attempts to improve on tree encodings in terms of both coverage and speed. With regard to coverage, the methods presented here both work for arbitrary finite meet semi-lattices. With regard to speed, they are both *flat-term encodings*, i.e., a term whose substructures are all reachable from the root by a path of length at most 1. Because first-order terms are of fixed arities, unification of a variable with a flat term can be compiled into a primitive recursive loop that compilers can unwind off-line. Unification of nested compound terms, however, typically involves some amount of pointer chasing on the heap before the actual addresses can be passed to the unifier.

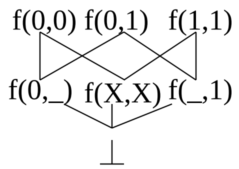

FIGURE 4. A flat-term encoding of Figure 3.

Thus, with all other things being equal, "broader beats deeper," i.e., two flat terms with n arguments can be unified more quickly than two terms of arity one, with subterms nested n levels deep. A tree encoding of a type hierarchy has terms whose depths are bounded by the maximum length of any subtype chain in the hierarchy, although with (limited) multiple or multi-dimensional inheritance, more than one path of that depth may be created; and in the case of true multiple inheritance (in the cases where the encoding works), some duplication of structure may be necessary at the ends of the paths. Flat-term encodings do not require redundant structure, but, in the *best* case, have arities equal to the length of the maximum subtype chain in the hierarchy — multiple inheritance can force it to be wider. In cases where both are applicable, it is an empirical question as to whether the speed-up from flatness outweighs the fact that very general types in a tree encoding have smaller term sizes.

Mellish (1991, 1992) proved (non-constructively) that while not all finite meet semi-lattices admit tree encodings, they do all admit flat-term encodings. While this encoding can instantiate arguments only to constants, it can also use individual variables in more than one argument position. A flat-term encoding of Figure 3 is given in Figure 4.

Mellish also showed (Mellish 1988) that what is now known as *Colmerauer's method* can be used to encode systemic networks using flat terms of arity $n+1$, where n is the number of possible property assignments allowed for by the network. The first argument is always 0, the last argument is always 1, and every assignment is assigned a pair of arguments in between. Every description that excludes an assignment numbered i is represented by a term whose ith and $i+1$st arguments are bound to a common term. If there are 5 possible assignments, then we use a term of arity 6 as follows (Mellish 1988):

```
        1       2       3       4       5
        |       |       |       |       |
f(0,    _,      _,      X,      X,      1)  (excludes assignment 4)
f(0,    0,      _,      X,      X,      1)  (excludes 1 and 4)
f(0,    X,      X,      X,      1,      1)  (excludes 2, 3, and 5)
```

The number of possible assignments is, in the worst case, exponential in the number of properties in the systemic network; so this method is not practical for encoding systemic networks, because it can yield terms with exponentially large arities.

Fall (1996, p. 94) observed that Colmerauer's method could be used for unification-preserving encodings of arbitrary ordered sets, but mistakenly assumed that the method yielded exponential-sized terms for that task as well. It is exponential for systemic networks because systemic networks can, in some cases, provide exponentially compact encodings of possible assignments. For the case of an enumerated set of assignments, the method yields linear-sized terms. Colmerauer, in fact, had originally used his method for computing arbitrary set intersections. ProFIT used this encoding in the same spirit for finite domains, i.e., flat type hierarchies with no appropriate features. We now consider a method that makes optimal use of Colmerauer's method with meet semi-lattices, a subcase of ordered sets.

2.2.1. *Method 1: Colmerauer's method for meet semi-lattices.* The first observation we can make is that meet semi-lattices can be decomposed into *modules*, pieces whose types can never unify:

DEFINITION 1. *Given a type hierarchy, $\langle T, \sqsubseteq\rangle$, the set of* modules *of $\langle T, \sqsubseteq \rangle$ is the finest partition of $T - \{\bot\}$, $M_1, \ldots M_m$, such that:*

1. *each M_i is upward-closed (w.r.t subsumption), and*
2. *if t_i and t_j are unifiable, then they belong to the same module*

Figure 1, for example, has three modules, one each rooted at *case*, *bool*, and *head*. In general, a module might not have a unique least type. Figure 3 has one module, for example. Modularizing in this fashion can be performed as the first step to any encoding strategy, because the modules can be encoded separately. The definition of module above can be generalized in order to develop hybrid tree-based / flat-term-based encodings as well, although for brevity, this will not be considered further here. Without loss of generality, we can now consider only type hierarchies that have one module with no least type. As in a tree encoding, $\bot$

can be represented by a Prolog variable; and if there is a least type in a module, then it can be represented by a term with a unique variable in every argument, i.e., the most general term of that functor and arity.

DEFINITION 2. *Given a type hierarchy, $\langle T, \sqsubseteq \rangle$, a type, $t \in T$, is* meet reducible *iff there are distinct types $u, v \in T$ not equal to t such that the meet of u and v, $u \sqcap v = t$ (and dually for* join reducible *types). The* meet irreducible *types of T are all types in T except the meet reducible types.*

PROPOSITION 1. *The meet irreducible types of a type hierarchy are precisely the maximally specific types and the types with one immediate subtype.*

PROPOSITION 2. *Every type hierarchy has a flat-term encoding with an arity equal to the number of meet irreducible types plus 1.*
Proof: Use Colmerauer's method on the set of meet irreducible types. Every type in the meet semi-lattice can be represented by the set of meet irreducible types that it subsumes. Maximally specific types only unify with themselves. All other types can then be characterized by the maximally specific types that they subsume except the types with one immediate subtype, which would be characterized by the same maximally specific subtypes as that immediate subtype. Those types are meet irreducible as well, however, and so are distinguished by their own occurrence in the representation.

One of the algorithms in Aït-Kaći, Boyer, Lincoln, and Nasr 1989 achieves essentially the same encoding, but for bit vectors. Finding the set of meet irreducible types takes at most cubic time — the time it takes to test all u and v for meets given a transitively closed adjacency representation of subsumption.

This method yields an optimal encoding in the sense that no encoding that uses Colmerauer's method can result in terms with a smaller arity than the number of meet irreducible types plus 1 (Fall 1996). Arity is the relevant measure since extra argument constants in the encoding come comparatively cheaply, and multiple occurrences of individual variables does not make unification slower. In general, however, this method does not yield an optimal flat-term encoding overall. Figure 5 shows a binary tree module, which has no join reducible types. No binary trees have types with only one immediate subtype, so the number of meet irreducible elements is the number of maximally specific types, or half the total number of types. It can be proven, however, that the flat-term encoding of smallest arity for a binary tree, in fact for any module without join reducible types, is equal to the length of its longest subtype chain,

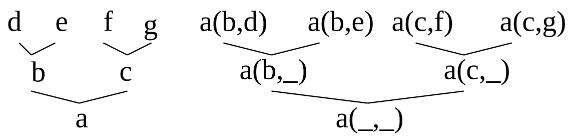

FIGURE 5. A binary tree and its optimal flat-term encoding.

which for binary trees, is equal to the logarithm of the total number of types. That encoding is shown in Figure 5.

What is the smallest arity required for arbitrary meet semi-lattices?

2.2.2. *Method 2: Parametrized Search for an Optimal Encoding.* A flat-term encoding is characterized by a choice of functor for the term (which is irrelevant, provided every module has one unique functor), a choice of arity of the term, and a choice of constants to instantiate some of the arguments of the term.

The set of first-order terms can be characterized as a meet semi-lattice in its own right, called the lattice of Generalized Atomic Formulae (GAF, Reynolds 1970). We only need the sublattice of flat terms, GAF_1; and for a given module, only those flat terms of the same principal functor (which can remain implicit) and arity, a, with argument constants, 0 through some k, $GAF^a_{1,k}$. This is finite. Subsumption in this sublattice should be familiar to anyone who has used Prolog — in $GAF^2_{1,1}$, for example $f(\_,\_)$ is the most general term, $f(\_,1) \sqsubseteq_{GAF} f(1,1)$, $f(1,\_) \sqsubseteq_{GAF} f(1,1)$ and the term with the same variable in both positions, $f(X,X) \sqsubseteq_{GAF} f(1,1)$.[3]

The fact that *every* finite meet semi-lattice has a Colmerauer-style encoding means that our choice of arity in a flat-term encoding never needs to be greater than the number of meet irreducible types of a module plus 1. There are other constraints on arity that can be proven as well (the proofs are omitted here for brevity):

DEFINITION 3. *Given type hierarchy, $\langle T, \sqsubseteq\rangle$ and type $t \in T$, the* information level *of t, $\delta_T(t)$, is the length of the longest subtype chain from $\perp$ to t.*

We can implicitly order our types into a sequence t_i, such that if $i < j$ then $\delta_T(t_i) \leq \delta_T(t_j)$, with $\perp$ as t_1. We can also extend δ to flat-term encodings, thinking of them as meet semi-lattices, $GAF^a_{1,k}$.

[3]We will not consider terms $f(X,Y)$ with $X \neq Y$, although Prologs with inequations would allow us to use these in encodings as well.

PROPOSITION 3. *Let $\bar{T}$ be a flat-term encoding of T and $\bar{t}$ be the term corresponding to $t \in T$ in $\bar{T}$. Then:*

1. *For any t, $\delta_{\bar{T}}(\bar{t}) \geq \delta_T(t)$.*
2. *If $\bar{t}_1 \sqcup \bar{t}_2 = \bar{t}_3$, then $\delta_{\bar{T}}(\bar{t}_3) = \delta_{\bar{T}}(\bar{t}_1) + \delta_{\bar{T}}(\bar{t}_2) - \delta_{\bar{T}}(\bar{t}_1 \sqcap \bar{t}_2) \leq \delta_{\bar{T}}(\bar{t}_1) + \delta_{\bar{T}}(\bar{t}_2)$.*
3. *For any t, its supertype branching factor $\sigma(t) \leq 2^{\delta_{\bar{T}}(\bar{t})} - 1$.*

The arity of the terms in a module's encoding must be constant, so (1) implies that that arity must be at least as large as the length of the longest subtype chain in T, since the type at the end of that chain must be encoded by a term of at least that arity. This lower bound can always be attained if there are no join reducible types. (2) says that the δ-value of the result of unification in a unification-preserving encoding cannot exceed the sum of its operands' δ-values — since we are not requiring our encodings to be meet-preserving, we might not know the value of $\delta_{\bar{T}}(\bar{t}_1 \sqcap \bar{t}_2)$. This means that join reducible types can, in general, force an encoding to have greater arity to allow for higher δ-values of two supertypes so that there will be enough "room" for their join. (3) essentially documents the same effect as a result of the bound on the number of terms that can possibly subsume a Prolog term of a given arity.

The choice of constants for instantiation of arguments is also bounded as a function of arity:

PROPOSITION 4. *In a type hierarchy, T, if there is a flat-term encoding of arity a, there is a flat-term encoding of arity a that uses no more than $a \cdot max(T)$ constants, where $max(T)$ is the number of maximally specific types in T.*

Proof: Because flat-term encodings preserve unification, they also preserve subsumption, so a constant used in any term is reflected in the same argument position of the encoding of some maximally specific type. There are $a \cdot \max(T)$ such positions. Of course, a Colmerauer-style encoding only uses two constants, 0 and 1.

The net result of these constraints is that there is a finite space of parameters — arity and number of constants — through which we can search for an optimal encoding, provided that we have a uniform representation of encodings through which to search. That representation can be achieved by looking at a different characterization of the algebras we are trying to encode, and relating them to the Prolog terms eligible to participate in our encodings.

	$\bot$	lax	cl	cng	lax&cl	lax&cng	cl&cng
$\bot$	1	1	1	1	1	1	1
lax	0	1	0	0	1	1	0
cl	0	0	1	0	1	0	1
cng	0	0	0	1	0	1	1
lax&cl	0	0	0	0	1	0	0
lax&cng	0	0	0	0	0	1	0
cl&cng	0	0	0	0	0	0	1

FIGURE 6. The subsumption matrix for Figure 3.

DEFINITION 4. *Given a type hierarchy,* $\langle T, \sqsubseteq \rangle$, *the* subsumption matrix *of* $\langle T, \sqsubseteq \rangle$, S, *is a* $|T| \times |T|$ *boolean matrix where* S_{ij} *is 1 if* $t_i \sqsubseteq t_j$.

Subsumption matrices can be computed by building a matrix of the reflexively closed immediate subsumption relation that a programmer defines, and then multiplying the matrix by itself until it reaches a fixed point, thus computing the transitive closure (Aït-Kaći, Boyer, Lincoln, and Nasr 1989). A subsumption matrix uniquely characterizes a finite meet semi-lattice's behavior with respect to unification, since joins are completely determined by subsumption. In fact, as observed in Aït-Kaći, Boyer, Lincoln, and Nasr 1989, each row can be used as an encoding of the type labelling t, with unification corresponding to component-wise AND. The subsumption matrix of Figure 3 is shown in Figure 6.

For a fixed a and k, $GAF^a_{1,k}$ has a fixed subsumption matrix. With this view of finite meet semi-lattices, finding a flat-term encoding of one with a subsumption matrix S amounts to finding the right rows and columns of a GAF lattice that will behave like S:

PROPOSITION 5. *Every flat-term encoding of a type hierarchy with subsumption matrix,* S, *uniquely corresponds to a selection matrix,* P, *such that:*

1. $P \cdot GAF^a_{1,k} \cdot P^{\mathsf{T}} = S$, *for some arity,* a, *some maximum constant,* k, *and*
2. *the rows (and columns) selected by* P *are closed under component-wise AND in* $GAF^a_{1,k}$.

An optimal flat-term encoding is one with the least arity a *for which such a* P *exists.*

The fact that we are using selection matrices — matrices with exactly one 1 in any row and no more than one 1 in any column, means that we satisfy the injectivity condition of Mellish 1991 (see Section 2.1, this paper). The first condition given here means that it satisfies Mellish's homomorphism condition — the terms corresponding to the rows and columns in $GAF^{a}_{1,k}$ preserve unification because they have the same subsumption matrix. The second condition is necessary to ensure Mellish's zero-preservation condition — it could be that we threw out the row of a term that is the unification of two terms that we included. There is no trace of it because we also threw out its column, and no trace appears in S if the two corresponding types were not unifiable. We want Prolog unification to fail when unification in T fails, however.

We can thus reduce the search for an encoding to a search for a selection matrix over a finite space of parameters: arity, ranging from 0 to the number of meet irreducible types, and number of constants, ranging from 0 to $a{\cdot}\max(T)$. S is no ordinary matrix, furthermore. If we break it into submatrices $A_{i,j}$ for rows of types with $\delta = i$ and columns of types with $\delta = j$, then for all i, j, $A_{i,i}$ is an identity matrix, and $A_{i,j}$ is a zero matrix when $i < j$. Due to the constraints mentioned above, we can consider the problem, to a great extent, independently by information level when solving for P. The general problem is, of course, still horrendously expensive. Even small parts of finding the optimal flat-term encoding are known to be NP-complete (Fall 1996).

3. Features

Once we add features, we need to accommodate their values in the encoding, including possibly circular structures, which reduce straightforwardly to circular Prolog terms. Carpenter (1992) proves that some type signatures inherently require dynamic type checking, which Prolog encodings, as defined here, cannot do. For most applications, only statically typable signatures are necessary, although it is possible to implement the non-statically typable ones using the functionality of an enhanced Prolog such as SICStus's `library(atts)`, which allows for hooks to unification. Using this library as a means of encoding TFSs is considered in the evaluation.

Tree-based encodings can add extra arguments at subterms where features are introduced. A TFS of type *noun*, from Figure 1, would be encoded as `head(subst(noun(case(nom)),plus,plus))`, where the two `plus` values are for the PRD and MOD features introduced by *head*. The logic of Carpenter 1992 allows subtypes to refine the value restrictions on features introduced by their supertypes, and for feature introduction at joins. ProFIT's declaration language allows for neither of these; but a tree encoding is compatible with them, in principle.

An alternative is to encode all of the feature values of a module as extra arguments at the top level of the subtype encoding. This again appeals to the wisdom, "broader beats deeper," particularly since feature values are themselves encoded TFSs. It has the additional advantage that binding a variable to a feature value, another very common operation, can in many cases be compiled out to a very efficient `arg/3` call in Prolog runtime code, where the tree-based encoding would require a more expensive term traversal. It also has the same empirical caveat as with subtype encoding — that empirical domains that make reference to a large number of TFSs with types more general than types that introduce features may still perform better with the tree encoding, because they avoid the extra unused feature positions.

How many extra argument positions do we need for a module's encoding? The naive answer is the number of features introduced in that module. It is possible to do better:

DEFINITION 5. *The* feature graph*, $G(M)$ of module M, is an undirected graph whose vertices correspond to the features introduced in M, and in which there is an edge, (F, G), iff F and G are appropriate to a common type in M.*

PROPOSITION 6. *The least number of argument positions required for the features of M in a flat encoding is the least N for which $G(M)$ is N-colorable.*

The positions correspond to the colors. This is related to using graph coloring in compiler design for register allocation. In Figure 1, the features PRD, MOD, and CASE form a graph that is at best 3-colorable, because they are all appropriate to the common type, *noun*.

4. EVALUATION

The alternatives presented here have been evaluated on the task of tabulation-based parsing with two English grammars using the same 11 sentences chosen on the basis of the size of their substring tables,[4] which loosely correlates with processing difficulty. Measurements were made on a dual-250-MHz SPARC Ultra 450 with 512 MB of RAM.

[4]The size ranges from 13 to 8551 edges with the HPSG grammar, and from 14 to 199 edges with the Bell Labs grammar.

The first, on which Figure 7 is based, is the HPSG grammar distributed with the ALE system, a straightforward encoding of the first five chapters of Pollard and Sag 1994, and a common benchmark for logic programming with TFSs. It has 162 types and 37 features, which decompose into a large number of small modules, each having at worst 5-colorable feature graphs. All modules but two are free of join reducible types, which means that they are optimally tree-encodable; but the two, lists and sets, are heavily used within the grammar. The top alternative depicted is the performance of a naive encoding of typed feature structures using the SICStus Prolog attributed variables library, where the type is represented as the value of an extra feature defined on every structure. The second uses the same library but with one attribute for every "color" of feature as described above, rather than for every feature. This takes advantage of the high modularity of the grammar. The third uses undocumented SICStus internal predicates to manipulate those attributes directly in order to exploit the existence of appropriateness conditions. The fourth is not a proper Prolog encoding — it uses a Prolog data structure that must be dereferenced before unification. This result was obtained from ALE 3.2. Both the third and the fourth use the exact feature arity of every type for its representation, so no coloring is needed. For non-statically typable modules, these two are the best alternatives available. The advantage of the third is that it can be used together with Prolog-term-encoded static modules because of the availability of the `verify_attributes/3` unification hook for attributed variables.

The last three are proper Prolog term encodings as elaborated upon here. The fifth was obtained from ProFIT 1.54 with its tree encoding method — the list and set modules are tree encodable. The last two use an optimal tree encoding on modules with no join reducible types. These are so easy to detect and the encoding is so quickly derived that no other choice makes sense. The sixth was obtained using the approximate method presented in Section 2.2.1 on the two other modules, but without feature graph coloring. The seventh uses the optimal method presented in Section 2.2.2 with feature coloring. The sixth and seventh bound the performance of the four possible permutations of encoding method with feature coloring; and, as can be seen, it makes very little difference. For this grammar, a completely polynomial approximation is worthwhile.

The second grammar (Figure 8) is a categorial grammar from Bell Labs encoded in typed feature logic, designed to have similar coverage to the ALE HPSG grammar. It has a total of 209 types and 27 features. It has only five modules, however, with the largest containing 119 of the types,

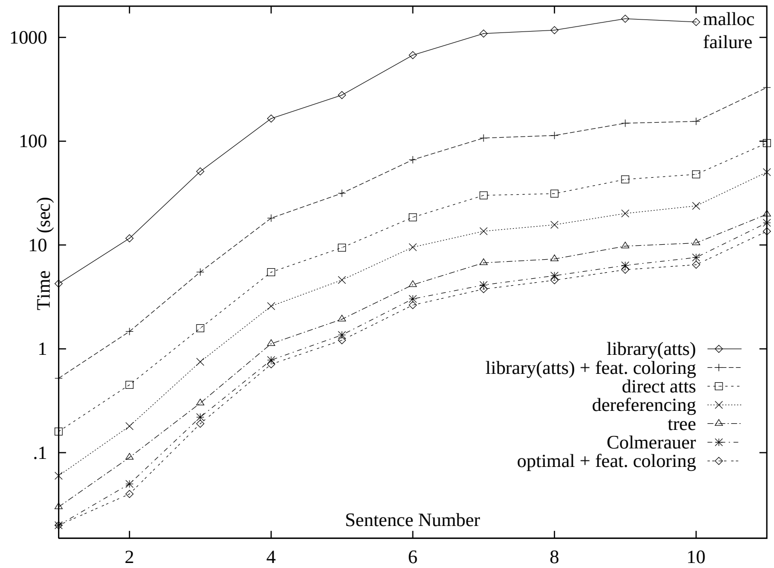

FIGURE 7. Evaluation on the ALE HPSG grammar.

including 88 meet-irreducible types, but having an optimal encoding of arity 6 — in fact, there is only one join-reducible type in it. Another module has 17 of the features, with a 6-colorable feature graph. The Bell Labs grammar is much faster than the HPSG grammar; but within its performance, feature coloring can be seen to be of more significance. Optimal type encoding is of great significance — even dereferencing is better than Colmerauer's method here.

5. Conclusion

Two methods have been presented to encode arbitrary type signatures as flat Prolog terms, which provides an improvement in speed over other general representation methods, and a competitive performance with tree encoding, in addition to its more general applicability. A few of the results, such as the graph-coloring reduction and the selection matrix reduction, are independently of some theoretical interest.

What remains now is to find heuristic methods that, for linguistically prevalent type signatures, can constrain the compile-time parametric

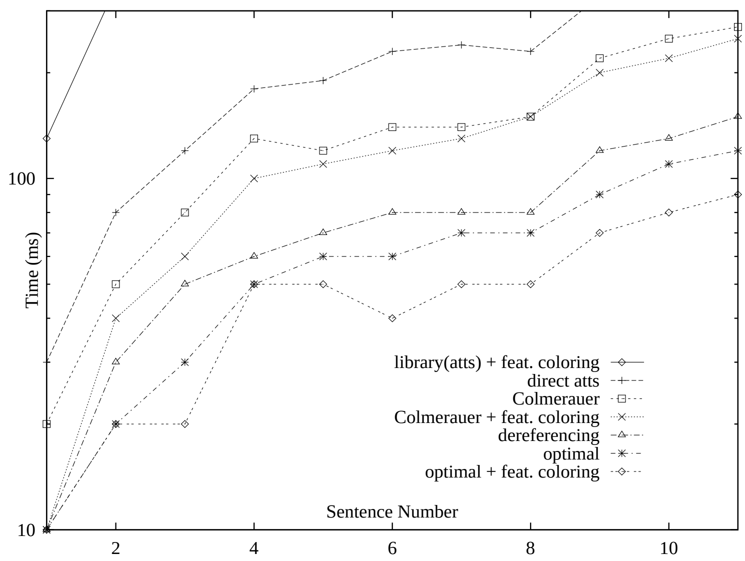

FIGURE 8. Evaluation on the Bell Labs Categorial Grammar.

search for optimal flat-term type encodings; and to conduct a wider empirical comparison than can be attempted in this paper to find hybrid encodings best suited to the realistic processing needs of the knowledge representation and linguistics communities.

REFERENCES

Aït-Kaći, H. (1993). An introduction to LIFE — programming with logic, inheritance, functions and equations. In D. Miller (Ed.), *Proceedings of the International Symposium on Logic Programming*, pp. 52–68. MIT Press.

Aït-Kaći, H., R. Boyer, P. Lincoln, and R. Nasr (1989). Efficient implementation of lattice operations. *ACM Transactions on Programming Languages and Systems 11*(1), 115–146.

Carpenter, B. (1992). *The Logic of Typed Feature Structures*. Cambridge.

Carpenter, B. and G. Penn (1996). Compiling typed attribute-value logic grammars. In H. Bunt and M. Tomita (Eds.), *Recent Advances in Parsing Technologies*. Kluwer.

Carpenter, B. and G. Penn (1998, September). *ALE 3.1 User's Manual*. Carnegie Mellon University. Available from the ALE Homepage: `http://www.sfs.nphil.uni-tuebingen.de/~gpenn/ale.html`.

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

Dahl, V. (1977). Un systeme deductif d'interrogation de banques de donnes en espagnol. Technical report, Groupe d'Intelligence Artificielle, Université de Marseille-Luminy.

Erbach, G. (1994). Multi-dimensional inheritance. In *Proceedings of KONVENS 94*. Springer.

Erbach, G. (1995). Profit: Prolog with features, inheritance and templates. In *Proceedings of EACL '95*.

Erbach, G. (1996). *Bottom-Up Earley Deduction for Preference-Driven Natural Language Processing*. Ph. D. thesis, Universität des Saarlandes.

Fall, A. (1996). *Reasoning with Taxonomies*. Ph. D. thesis, Simon Fraser University.

Gerdemann, D. (1995). Term encoding of typed feature structures. In *Proceedings of the 4th International Workshop on Parsing Technologies*, pp. 89–97.

Hudson, R. A. (1981). Systemic generative grammar. In M. A. K. Halliday and J. R. Martin (Eds.), *Readings in Systemic Linguistics*. Batsford Academic.

Makino, T., K. Torisawa, and J. Tsuji (1998). LiLFeS — practical unification-based programming system for typed feature structures. In *COLING/ACL '98*.

Mellish, C. (1988). Implementing systemic classification by unification. *Computational Linguistics 14*(1), 40–51.

Mellish, C. (1991). Graph-encodable description spaces. Technical report, University of Edinburgh. DYANA Deliverable R3.2B.

Mellish, C. (1992). Term-encodable description spaces. In D. Brough (Ed.), *Logic Programming: New Frontiers*, pp. 189–207. Kluwer.

Meyer, R., B. Dumant, and P. Van Roy (1994). The Half_Life 0.1 system. Technical report, DEC Paris Research Laboratory. Available from `http://www.info.ucl.ac.be/people/PVR/halflife.html`.

Pollard, C. and I. Sag (1994). *Head-driven Phrase Structure Grammar*. Chicago.

Reynolds, J. (1970). Transformational systems and the algebraic structure of atomic formulas. In B. Meltzer and D. Michie (Eds.), *Machine Intelligence 5*. Edinburgh.

Simpkins, N. and M. Groenendijk (1994, August). Multiple inheritance. Technical Report ALEP-1.3, Cray Systems.

Van Roy, P., M. Mehl, and R. Scheidhauer (1996). Integrating efficient records into concurrent constraint programming. In *8th International Symposium on Programming Languages, Implementations, Logics, and Programs (PLILP 96)*, pp. 438–453.

Wintner, S. (1997). *An Abstract Machine for Unification Grammars with Applications to an HPSG Grammar for Hebrew*. Ph. D. thesis, Technion.