Generalized Encoding of Description Spaces and its Application to Typed Feature Structures

Gerald Penn
Department of Computer Science
University of Toronto
10 King's College Rd.
Toronto M5S 3G4, Canada

Abstract

This paper presents a new formalization of a unification- or join-preserving encoding of partially ordered sets that more essentially captures what it means for an encoding to preserve joins, generalizing the standard definition in AI research. It then shows that every statically typable ontology in the logic of typed feature structures can be encoded in a data structure of fixed size without the need for resizing or additional union-find operations. This is important for any grammar implementation or development system based on typed feature structures, as it significantly reduces the overhead of memory management and reference-pointer-chasing during unification.

1 Motivation

The logic of typed feature structures (Carpenter, 1992) has been widely used as a means of formalizing and developing natural language grammars that support computationally efficient parsing, generation and SLD resolution, notably grammars within the Head-driven Phrase Structure Grammar (HPSG) framework, as evidenced by the recent successful development of the LinGO reference grammar for English (LinGO, 1999). These grammars are formulated over a finite vocabulary of features and partially ordered types, in respect of constraints called *appropriateness conditions*. Appropriateness specifies, for each type, all and only the features that take values in feature structures of that type, along with

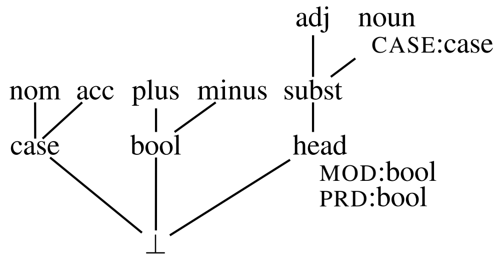

Figure 1: A sample type system with appropriateness conditions.

the types of values (*value restrictions*) those feature values must have. In Figure 1,[1] for example, all *head*-typed TFSs must have *bool*-typed values for the features MOD and PRD, and no values for any other feature.

Relative to data structures like arrays or logical terms, typed feature structures (TFSs) can be regarded as an expressive refinement in two different ways. First, they are typed, and the type system allows for subtyping chains of unbounded depth. Figure 1 has a chain of length 4 from $\perp$ to *noun*. Pointers to arrays and logical terms can only monotonically "refine" their (syntactic) type from unbound (for logical terms, variables) to bound. Second, although all the TFSs of a given type have the same features because of appropriateness, a TFS may acquire more features when it promotes to a subtype. If a *head*-typed TFS promotes to *noun* in the type system above, for example, it acquires one extra *case*-valued feature, CASE. When a subtype has two or

[1] In this paper, Carpenter's (1992) convention of using $\perp$ as the most general type, and depicting subtypes above their supertypes is used.

more incomparable supertypes, a TFS can also multiply inherit features from other supertypes when it promotes.

The overwhelmingly most prevalent operation when working with TFS-based grammars is unification, which corresponds mathematically to finding a least upper bound or *join*. The most common instance of unification is the special case in which a TFS is unified with the most general TFS that satisfies a description stated in the grammar. This special case can be decomposed at compile-time into more atomic operations that (1) promote a type to a subtype, (2) bind a variable, or (3) traverse a feature path, according to the structure of the description.

TFSs actually possess most of the properties of fixed-arity terms when it comes to unification, due to appropriateness. Nevertheless, unbounded subtyping chains and acquiring new features conspire to force most internal representations of TFSs to perform extra work when promoting a type to a subtype to earn the expressive power they confer. Upon being repeatedly promoted to new subtypes, they must be repeatedly resized or repeatedly referenced with a pointer to newly allocated representations, both of which compromise locality of reference in memory and/or involve pointer-chasing. These costs are significant.

Because appropriateness involves value restrictions, simply padding a representation with some extra space for future features at the outset must guarantee a proper means of filling that extra space with the right value when it is used. Internal representations that lazily fill in structure must also be wary of the common practice in description languages of binding a variable to a feature value with a scope larger than a single TFS — for example, in sharing structure between a daughter category and a mother category in a phrase structure rule. In this case, the representation of a feature's value must also be interpretable independent of its context, because two separate TFSs may refer to that variable.

These problems are artifacts of not using a representation which possesses what in knowledge representation is known as a *join-preserving encoding* of a grammar's TFSs — in other words, a representation with an operation that naturally behaves like TFS-unification. The next section presents the standard definition of join-preserving encodings and provides a generalization that more essentially captures what it means for an encoding to preserve joins. Section 3 formalizes some of the defining characteristics of TFSs as they are used in computational linguistics. Section 4 shows that these characteristics quite fortuitously agree with what is required to guarantee the existence of a join-preserving encoding of TFSs that needs no resizing or extra referencing during type promotion. Section 5 then shows that a generalized encoding exists in which variable-binding scope can be larger than a single TFS — a property no classical encoding has.

Earlier work on graph unification has focussed on labelled graphs with no appropriateness, so the central concern was simply to minimize structure copying. While this is clearly germane to TFSs, appropriateness creates a tradeoff among copying, the potential for more compact representations, and other memory management issues such as locality of reference that can only be optimized empirically and relative to a given grammar and corpus (a recent example of which can be found in Callmeier (2001)). While the present work is a more theoretical consideration of how unification in one domain can simulate unification in another, the data structure described here is very much motivated by the encoding of TFSs as Prolog terms allocated on a contiguous WAM-style heap. In that context, the emphasis on fixed arity is really an attempt to avoid copying, and lazily filling in structure is an attempt to make encodings compact, but only to the extent that join preservation is not disturbed. While this compromise solution must eventually be tested on larger and more diverse grammars, it has been shown to reduce the total parsing time of a large corpus on the ALE HPSG benchmark grammar of English (Penn, 1993) by a factor of about 4 (Penn, 1999).

2 Join-Preserving Encodings

We may begin with a familiar definition from discrete mathematics:

Definition 1 *Given two partial orders* $\langle P, \sqsubseteq_P \rangle$ *and* $\langle R, \sqsubseteq_R \rangle$*, a function* $f : P \longrightarrow R$ *is an* order-embedding *iff, for every* $x, y \in P$, $x \sqsubseteq_P y$ *iff* $f(x) \sqsubseteq_R f(y)$.

An order-embedding preserves the behavior of the order relation (for TFS type systems, subtyping;

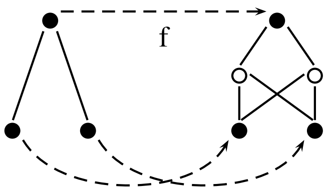

Figure 2: An example order-embedding that cannot translate least upper bounds.

for TFSs themselves, subsumption) in the encoding codomain. As shown in Figure 2, however, order embeddings do not always preserve operations such as least upper bounds. The reason is that the image of f may not be closed under those operations in the codomain. In fact, the codomain could provide joins where none were supposed to exist, or, as in Figure 2, no joins where one was supposed to exist. Mellish (1991; 1992) was the first to formulate join-preserving encodings correctly, by explicitly requiring this preservation. Let us write $p \sqcup_P q$ for the join of p and q in partial order P.

Definition 2 *A partial order* $\langle P, \sqsubseteq \rangle$ *is* bounded complete (BCPO) *iff every set of elements with a common upper bound has a least upper bound.*

Bounded completeness ensures that unification or joins are well-defined among consistent types.

Definition 3 *Given two BCPOs,* P *and* R*,* $f : P \longrightarrow R$ *is a* classical join-preserving encoding of P into R *iff:*

- **injectivity** f *is an injection,*
- **zero preservation** $f(p \sqcup_P q)\uparrow$[2] *iff* $f(p) \sqcup_R f(q)\uparrow$*, and*
- **join homomorphism** $f(p \sqcup_P q) = f(p) \sqcup_R f(q)$*, where they exist.*

Join-preserving encodings are automatically order-embeddings because $p \sqcup q = q$ iff $p \sqsubseteq q$.

There is actually a more general definition:

Definition 4 *Given two BCPOs,* P *and* R*,* $f : P \longrightarrow Pow(R)$ *is a* (generalized) join-preserving encoding of P into R *iff:*

- **totality** *for all* $p \in P$*,* $f(p) \neq \emptyset$*,*
- **disjointness** $f(p) \cap f(q) \neq \emptyset$ *iff* $p = q$*,*

[2]We use the notation $f(\vec{x})\uparrow$ to mean $f(\vec{x})$ is undefined, and $f(\vec{x})\downarrow$ to mean $f(\vec{x})$ is defined.

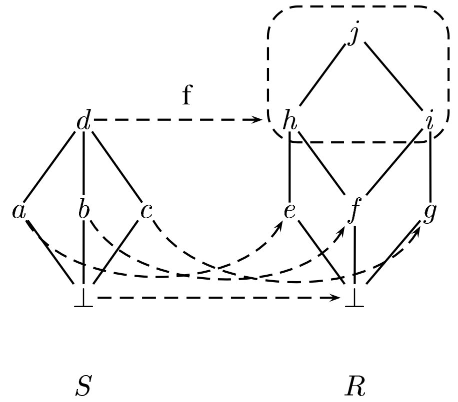

Figure 3: A non-classical join-preserving encoding between BCPOs for which no classical join-preserving encoding exists.

- **zero preservation** *for all* $\bar{p} \in f(p)$ *and* $\bar{q} \in f(q)$*,* $p \sqcup_P q\uparrow$ *iff* $\bar{p} \sqcup_R \bar{q}\uparrow$*, and*
- **join homomorphism** *for all* $\bar{p} \in f(p)$ *and* $\bar{q} \in f(q)$*,* $\bar{p} \sqcup_R \bar{q} \in f(p \sqcup_P q)$*, where they exist.*

When f maps elements of P to singleton sets in R, then f reduces to a classical join-preserving encoding. It is not necessary, however, to require that only one element of R represent an element of P, provided that it does not matter which representative we choose at any given time. Figure 3 shows a generalized join-preserving encoding between two partial orders for which no classical encoding exists. There is no classical encoding of S into R because no three elements can be found in R that pairwise unify to a common join. A generalized encoding exists because we can choose three potential representatives for d: one (h) for unifying the representatives of a and b, one (i) for unifying the representatives of b and c, and one (j) for unifying the representatives of a and c. Notice that the set of representatives for d must be closed under unification.

Although space does not permit here, this generalization has been used to prove that well-typing, an alternative interpretation of appropriateness, is equivalent in its expressive power to the interpretation used here (called total well-typing; Carpenter, 1992); that *multi-dimensional inheritance* (Erbach, 1994) adds no expressive power to any TFS type system; that TFS type systems can encode systemic networks in polynomial space using *extensional types* (Carpenter, 1992); and that certain uses of parametric typing with TFSs also add no expressive power to the type system (Penn, 2000).

3 TFS Type Systems

There are only a few common-sense restrictions we need to place on our type systems:

Definition 5 *A* TFS type system *consists of a finite BCPO of types,* $\langle P, \sqsubseteq \rangle$*, a finite set of features Feat, and a partial function,* $Approp : Feat \times T \longrightarrow T$ *such that, for every* $\text{F} \in Feat$*:*

- ***(Feature Introduction)*** *there is a type* $Intro(\text{F}) \in T$ *such that:* $Approp(\text{F}, Intro(\text{F}))\downarrow$*, and for all* $t \in T$*, if* $Approp(\text{F}, t)\downarrow$*, then* $Intro(\text{F}) \sqsubseteq t$*, and*
- ***(Upward Closure / Right Monotonicity)*** *if* $Approp(\text{F}, s)\downarrow$ *and* $s \sqsubseteq t$*, then* $Approp(\text{F}, t)\downarrow$ *and* $Approp(\text{F}, s) \sqsubseteq Approp(\text{F}, t)$*.*

The function *Approp* maps a feature and type to the value restriction on that feature when it is appropriate to that type. If it is not appropriate, then *Approp* is undefined at that pair. Feature introduction ensures that every feature has a least type to which it is appropriate. This makes description compilation more efficient. Upward closure ensures that subtypes inherit their supertypes' features, and with consistent value restrictions. The combination of these two properties allows us to annotate a BCPO of types with features and value restrictions only where the feature is introduced or the value restriction is refined, as in Figure 1.

A very useful property for type systems to have is *static typability*. This means that if two TFSs that are well-formed according to appropriateness are unifiable, then their unification is automatically well-formed as well — no additional work is necessary.

Theorem 1 (Carpenter, 1992) *An appropriateness specification is statically typable iff, for all types* s, t *such that* $s \sqcup t\downarrow$*, and all* $\text{F} \in Feat$*:*

$$Approp(\text{F}, s \sqcup t) = \begin{cases} Approp(\text{F}, s) \sqcup Approp(\text{F}, t) & \textit{if } Approp(\text{F}, s)\downarrow \textit{ and } Approp(\text{F}, t)\downarrow \\ Approp(\text{F}, s) & \textit{if only } Approp(\text{F}, s)\downarrow \\ Approp(\text{F}, t) & \textit{if only } Approp(\text{F}, t)\downarrow \\ \textit{unrestricted} & \textit{otherwise} \end{cases}$$

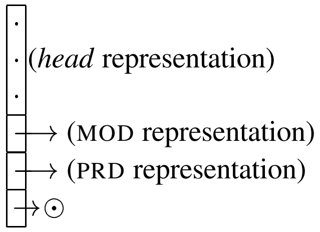

Figure 4: A fixed array representation of the TFS in Figure 5.

$$\begin{bmatrix} \textit{head} & \\ \text{MOD} & \textit{plus} \\ \text{PRD} & \textit{plus} \end{bmatrix}$$

Figure 5: A TFS of type *head* from the type system in Figure 1.

Not all type systems are statically typable, but a type system can be transformed into an equivalent statically typable type system plus a set of universal constraints, the proof of which is omitted here. In linguistic applications, we normally have a set of universal constraints anyway for encoding principles of grammar, so it is easy and computationally inexpensive to conduct this transformation.

4 Static Encodability

As mentioned in Section 1, what we want is an encoding of TFSs with a notion of unification that naturally corresponds to TFS-unification. As discussed in Section 3, static typability is something we can reasonably guarantee in our type systems, and is therefore something we expect to be reflected in our encodings — no extra work should be done apart from combining the types and recursing on feature values. If we can ensure this, then we have avoided the extra work that comes with resizing or unnecessary referencing and pointer-chasing.

As mentioned above, what would be best from the standpoint of memory management is simply a fixed array of memory cells, padded with extra space to accommodate features that might later be added. We will call these *frames*. Figure 4 depicts a frame for the *head*-typed TFS in Figure 5. In a frame, the representation of the type can either be (1) a bit vector encoding the type,[3] or (2) a reference pointer

[3]Instead of a bit vector, we could also use an index into a table if least upper bounds are computed by table look-up.

to another frame. If backtracking is supported in search, changes to the type representation must be trailed. For each appropriate feature, there is also a pointer to a frame for that feature's value. There are also additional pointers for future features (for *head*, CASE) that are grounded to some distinguished value indicating that they are unused — usually a circular reference to the referring array position. Cyclic TFSs, if they are supported, would be represented with cyclic (but not 1-cyclic) chains of pointers.

Frames can be implemented either directly as arrays, or as Prolog terms. In Prolog, the type representation could either be a term-encoding of the type, which is guaranteed to exist for any finite BCPO (Mellish, 1991; Mellish, 1992), or in extended Prologs, another trailable representation such as a mutable term (Aggoun and Beldiceanu, 1990) or an attributed value (Holzbaur, 1992). Padding the representation with extra space means using a Prolog term with extra arity. A distinguished value for unused arguments must then be a unique unbound variable.[4]

4.1 Restricting the Size of Frames

At first blush, the prospect of adding as many extra slots to a frame as there could be extra features in a TFS sounds hopelessly unscalable to large grammars. While recent experience with LinGO (1999) suggests a trend towards modest increases in numbers of features compared to massive increases in numbers of types as grammars grow large, this is nevertheless an important issue to address. There are two discrete methods that can be used in combination to reduce the required number of extra slots:

Definition 6 *Given a finite BCPO, $\langle P, \sqsubseteq \rangle$, the set of* modules *of $\langle P, \sqsubseteq \rangle$ is the finest partition of $P \backslash \{\bot\}$, $M_1, \ldots M_m$, such that (1) each M_i is upward-closed (with respect to subtyping), and (2) if two types have a least upper bound, then they belong to the same module.*

Trivially, if a feature is introduced at a type in one module, then it is not appropriate to any type in any other module. As a result, a frame for a TFS only needs to allow for the features appropriate to the module of its type. Even this number can normally be reduced:

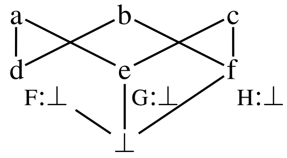

Figure 6: A type system with three features and a three-colorable feature graph.

Definition 7 *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 maximally specific type in M.*

Proposition 1 *The least number of feature slots required for a frame of any type in M is the least N for which $G(M)$ is N-colorable.*

There are type systems, of course, for which modularization and graph-coloring will not help. Figure 6, for example, has one module, three features, and a three-clique for a feature graph. There are statistical refinements that one could additionally make, such as determining the empirical probability that a particular feature will be acquired and electing to pay the cost of resizing or referencing for improbable features in exchange for smaller frames.

4.2 Correctness of Frames

With the exception of extra slots for unused feature values, frames are clearly isomorphic in their structure to the TFSs they represent. The implementation of unification that we prefer to avoid resizing and referencing is to (1) find the least upper bound of the types of the frames being unified, (2) update one frame's type to the least upper bound, and point the other's type representation to it, and (3) recurse on respective pairs of feature values. The frame does not need to be resized, only the types need to be referenced, and in the special case of promoting the type of a single TFS to a subtype, the type only needs to be trailed. If cyclic TFSs are not supported, then acyclicity must also be enforced with an occurs-check.

The correctness of frames as a join-preserving encoding of TFSs thus depends on being able to make sense of the values in these unused positions. The

[4]Prolog terms require one additional unbound variable per TFS (sub)term in order to preserve the intensionality of the logic — unlike Prolog terms, structurally identical TFS substructures are not identical unless explicitly structure-shared.

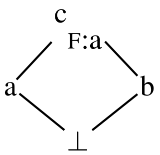

Figure 7: A type system that introduces a feature at a join-reducible type.

$$\begin{bmatrix} \textit{head} & \\ \textsc{mod} & \textit{plus} \\ \textsc{prd} & \textit{bool} \end{bmatrix}$$

Figure 8: A TFS of type *head* in which one feature value is a most general satisfier of its feature's value restriction.

problem is that features may be introduced at join-reducible types, as in Figure 7. There is only one module, so the frames for *a* and *b* must have a slot available for the feature F. When an *a*-typed TFS unifies with a *b*-typed TFS, the result will be of type *c*, so leaving the slot marked unused after recursion would be incorrect — we would need to look in a table to see what value to assign it. An alternative would be to place that value in the frames for *a* and *b* from the beginning. But since the value itself must be of type *a* in the case of Figure 7, this strategy would not yield a finite representation.

The answer to this conundrum is to use a distinguished circular reference in a slot iff the slot is either unused *or* the value it contains is (1) the most general satisfier of the value restriction of the feature it represents and (2) not structure-shared with any other feature in the TFS.[5] During unification, if one TFS is a circular reference, and the other is not, the circular reference is referenced to the other. If both values are circular references, then one is referenced to the other, which remains circular. The feature structure in Figure 8, for example, has the frame representation shown in Figure 9. The PRD value is a TFS of type *bool*, and this value is not shared with any other structure in the TFS. If the values of MOD and PRD are both *bool*-typed, then if they are shared (Figure 10), they do not use circu-

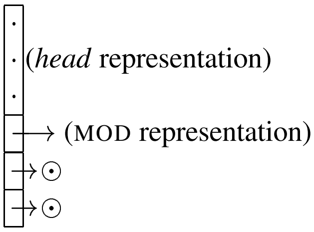

Figure 9: The frame for Figure 8.

$$\begin{bmatrix} \textit{head} & \\ \textsc{mod} & \boxed{1}\ \textit{bool} \\ \textsc{prd} & \boxed{1} \end{bmatrix}$$

Figure 10: A TFS of type *head* in which both feature values are most general satisfiers of the value restrictions, but they are shared.

lar references (Figure 11), and if they are not shared (Figure 12), both of them use a different circular reference (Figure 13).

With this convention for circular references, frames are a classical join-preserving encoding of the TFSs of any statically typable type system. Although space does not permit a complete proof here, the intuition is that (1) most general satisfiers of value restrictions necessarily subsume every other value that a totally well-typed TFS could take at that feature, and (2) when features are introduced, their initial values are not structure-shared with any other substructure. Static typability ensures that value restrictions unify to yield value restrictions, except in the final case of Theorem 1. The following lemma deals with this case:

Lemma 1 *If Approp is statically typable,* $s \sqcup t\downarrow$, *and for some* $\textsc{f} \in$ *Feat,* $Approp(\textsc{f}, s)\uparrow$ *and* $Approp(\textsc{f}, t)\uparrow$, *then either* $Approp(\textsc{f}, s \sqcup t)\uparrow$ *or*

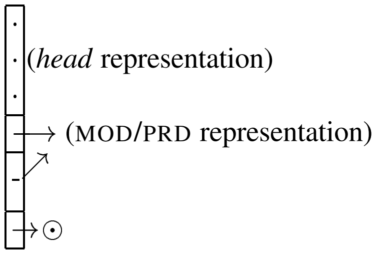

Figure 11: The frame for Figure 10.

[5]The sole exception is a TFS of type $\perp$, which by definition belongs to no module and has no features. Its representation is a distinguished circular reference, unless two or more feature values share a single $\perp$-typed TFS value, in which case one is a circular reference and the rest point to it. The circular one can be chosen canonically to ensure that the encoding is still classical.

$$\begin{bmatrix} \textit{head} & \\ \text{MOD} & \textit{bool} \\ \text{PRD} & \textit{bool} \end{bmatrix}$$

Figure 12: A TFS of type *head* in which both feature values are most general satisfiers of the value restrictions, and they are not shared.

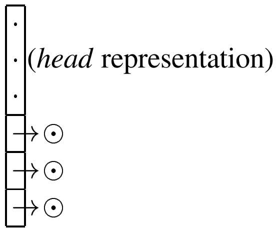

Figure 13: The frame for Figure 12.

$Approp(\text{F}, s \sqcup t) = Approp(\text{F}, Intro(\text{F}))$.

Proof: Suppose $Approp(\text{F}, s \sqcup t)\downarrow$. Then $Intro(\text{F}) \sqsubseteq s \sqcup t$. $Approp(\text{F}, s)\uparrow$ and $Approp(\text{F}, t)\uparrow$, so $Intro(\text{F}) \not\sqsubseteq s$ and $Intro(\text{F}) \not\sqsubseteq t$. So there are three cases to consider:

***Intro*(F) $=$ s $\sqcup$ t**: then the result trivially holds.

s $\sqsubseteq$ *Intro*(F) but t $\not\sqsubseteq$ *Intro*(F) (or by symmetry, the opposite): then we have the situation in Figure 14. It must be that $Intro(\text{F}) \sqcup t = s \sqcup t$, so by static typability, the lemma holds.

s $\not\sqsubseteq$ *Intro*(F) and t $\not\sqsubseteq$ *Intro*(F): $s \sqsubseteq s \sqcup t$ and $Intro(\text{F}) \sqsubseteq s \sqcup t$, so s and $Intro(\text{F})$ are consistent. By bounded completeness, $s \sqcup Intro(\text{F})\downarrow$ and $s \sqcup Intro(\text{F}) \sqsubseteq s \sqcup t$. By upward closure, $Approp(\text{F}, Intro(\text{F}) \sqcup s)\downarrow$ and by static typability, $Approp(\text{F}, Intro(\text{F}) \sqcup s) = Approp(\text{F}, Intro(\text{F}))$. Furthermore, $(Intro(\text{F}) \sqcup s) \sqcup t = s \sqcup t$; thus by static typability the lemma holds. $\square$

This lemma is very significant in its own right — it says that we know more than Carpenter's Theorem 1. An introduced feature's value restriction can always be predicted in a statically typable type system. The lemma implicitly relies on feature intro-

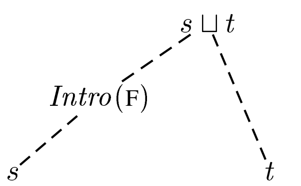

Figure 14: The second case in the proof of Lemma 1.

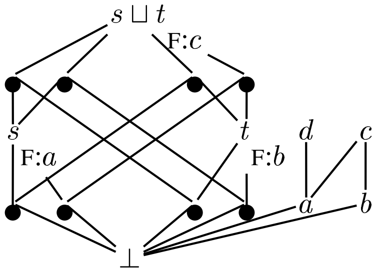

Figure 15: A statically typable "type system" that multiply introduces F at join-reducible elements with different value restrictions.

duction, but in fact, the result holds if we allow for multiple introducing types, provided that all of them agree on what the value restriction for the feature should be. Would-be type systems that multiply introduce a feature at join-reducible elements (thus requiring some kind of distinguished-value encoding), disagree on the value restriction, and still remain statically typable are rather difficult to come by, but they do exist, and for them, a frame encoding will not work. Figure 15 shows one such example. In this signature, the unification:

$$\begin{bmatrix} \text{s} & \\ \text{F} & \text{d} \end{bmatrix} \sqcup \begin{bmatrix} \text{t} & \\ \text{F} & \text{b} \end{bmatrix} \uparrow$$

does not exist, but the unification of their frame encodings must succeed because the t-typed TFS's F value must be encoded as a circular reference. To the best of the author's knowledge, there is no fixed-size encoding for Figure 15.

5 Generalized Term Encoding

In practice, this classical encoding is not good for much. Description languages typically need to bind variables to various substructures of a TFS, F, and then pass those variables outside the substructures of F where they can be used to instantiate the value of another feature structure's feature, or as arguments to some function call or procedural goal. If a value in a single frame is a circular reference, we can properly understand what that reference encodes with the above convention by looking at its context, i.e., the type. Outside the scope of that frame, we have no way of knowing which feature's value restriction it is supposed to encode.

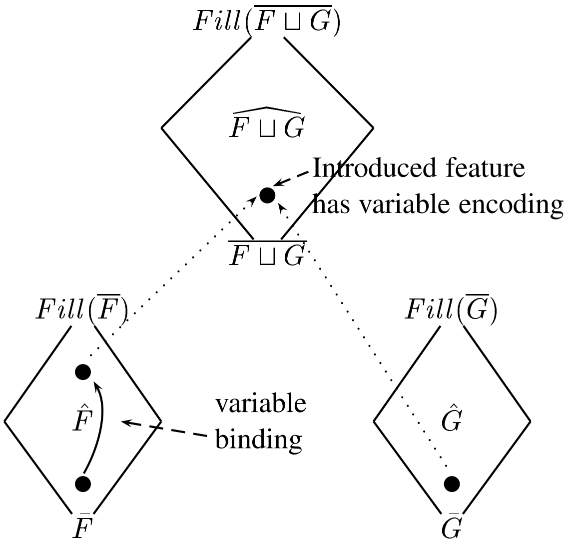

Figure 16: A pictorial overview of the generalized encoding.

A generalized term encoding provides an elegant solution to this problem. When a variable is bound to a substructure that is a circular reference, it can be filled in with a frame for the most general satisfier that it represents and then passed out of context. Having more than one representative for the original TFS is consistent, because the set of representatives is closed under this filling operation.

A schematic overview of the generalized encoding is in Figure 16. Every set of frames that encode a particular TFS has a least element, in which circular references are always opted for as introduced feature values. This is the same element as the classical encoding. It also has a greatest element, in which every unused slot still has a circular reference, but all unshared most general satisfiers are filled in with frames. Whenever we bind a variable to a substructure of a TFS, filling pushes the TFS's encoding up within the same set to some other encoding. As a result, at any given point in time during a computation, we do not exactly know which encoding we are using to represent a given TFS. Furthermore, when two TFSs are unified successfully, we do not know exactly what the result will be, but we do know that it falls inside the correct set of representatives because there is at least one frame with circular references for the values of every newly introduced feature.

6 Conclusion

Simple frames with extra slots and a convention for filling in feature values provide a join-preserving encoding of any statically typable type system, with no resizing and no referencing beyond that of type representations. A frame thus remains stationary in memory once it is allocated. A generalized encoding, moreover, is robust to side-effects such as extra-logical variable-sharing. Frames have many potential implementations, including Prolog terms, WAM-style heap frames, or fixed-sized records.

References

A. Aggoun and N. Beldiceanu. 1990. Time stamp techniques for the trailed data in constraint logic programming systems. In S. Bourgault and M. Dincbas, editors, *Programmation en Logique, Actes du 8eme Seminaire*, pages 487–509.

U. Callmeier. 2001. Efficient parsing with large-scale unification grammars. Master's thesis, Universitaet des Saarlandes.

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

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

C. Holzbaur. 1992. Metastructures vs. attributed variables in the context of extensible unification. In M. Bruynooghe and M. Wirsing, editors, *Programming Language Implementation and Logic Programming*, pages 260–268. Springer Verlag.

LinGO. 1999. The LinGO grammar and lexicon. Available on-line at **http://lingo.stanford.edu**.

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

C. Mellish. 1992. Term-encodable description spaces. In D.R. Brough, editor, *Logic Programming: New Frontiers*, pages 189–207. Kluwer.

G. Penn. 1993. The ALE HPSG benchmark grammar. Available on-line at **http://www.cs.toronto.edu/~gpenn/ale.html**.

G. Penn. 1999. An optimized Prolog encoding of typed feature structures. In *Proceedings of the 16th International Conference on Logic Programming (ICLP-99)*, pages 124–138.

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