Tractability and Structural Closures in Attribute Logic Type Signatures

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

Abstract

This paper considers three assumptions conventionally made about signatures in typed feature logic that are in potential disagreement with current practice among grammar developers and linguists working within feature-based frameworks such as HPSG: meet-semi-latticehood, unique feature introduction, and the absence of subtype covering. It also discusses the conditions under which each of these can be tractably restored in realistic grammar signatures where they do not already exist.

1 Introduction

The logic of typed feature structures (LTFS, Carpenter 1992) and, in particular, its implementation in the Attribute Logic Engine (ALE, Carpenter and Penn 1996), have been widely used as a means of formalising and developing grammars of natural languages that support computationally efficient parsing and SLD resolution, notably grammars within the framework of Head-driven Phrase Structure Grammar (HPSG, Pollard and Sag 1994). These grammars are formulated using a vocabulary provided by a finite partially ordered set of types and a set of features that must be specified for each grammar, and feature structures in these grammars must respect certain constraints that are also specified. These include *appropriateness conditions*, which specify, for each type, all and only the features that take values in feature structures of that type, and with which types of values (value restrictions). There are also more general implicational constraints of the form $\tau \rightarrow \phi$, where τ is a type, and ϕ is an expression from LTFS's description language. In LTFS and ALE, these four components, a partial order of types, a set of features, appropriateness declarations and type-antecedent constraints can be taken as the *signature* of a grammar, relative to which descriptions can be interpreted.

LTFS and ALE also make several assumptions about the structure and interpretation of this partial order of types and about appropriateness, some for the sake of generality, others for the sake of efficiency or simplicity. Appropriateness is generally accepted as a good thing, from the standpoints of both efficiency and representational accuracy, and while many have advocated the need for implicational constraints that are even more general, type-antecedent constraints at the very least are also accepted as being necessary and convenient. Not all of the other assumptions are universally observed by formal linguists or grammar developers, however.

This paper addresses the three most contentious assumptions that LTFS and ALE make, and how to deal with their absence in a tractable manner. They are:

1. **Meet-semi-latticehood:** every partial order of types must be a meet semi-lattice. This implies that every consistent pair of types has a least upper bound.

2. **Unique feature introduction:** for every feature, F, there is a unique most general type to which F is appropriate.

3. **No subtype covering:** there can be feature structures of a non-maximally-specific type that are not typable as any of its maximally specific subtypes. When subtype covering is not assumed, feature structures themselves

can be partially ordered and taken to represent partial information states about some set of objects. When subtype covering is assumed, feature structures are discretely ordered and totally informative, and can be taken to represent objects in the (linguistic) world themselves. The latter interpretation is subscribed to by Pollard and Sag (1994), for example.

All three of these conditions have been claimed elsewhere to be either intractable or impossible to restore in grammar signatures where they do not already exist. It will be argued here that: (1) restoring meet-semi-latticehood is theoretically intractable, for which the worst case bears a disquieting resemblance to actual practice in current large-scale grammar signatures, but nevertheless can be efficiently compilable in practice due to the sparseness of consistent types; (2) unique feature introduction can always be restored to a signature in low-degree polynomial time, and (3) while type inferencing when subtype covering is assumed is intractable in the worst case, a very elegant constraint logic programming solution combined with a special compilation method exists that can restore tractability in many practical contexts. Some simple completion algorithms and a corrected NP-completeness proof for non-disjunctive type inferencing with subtype covering are also provided.

2 Meet-semi-latticehood

In LTFS and ALE, partial orders of types are assumed to be meet semi-lattices:

Definition 1 *A partial order,* $\langle P, \sqsubseteq \rangle$, *is a* meet semi-lattice *iff for any* $x, y \in P$, $x \sqcap y\downarrow$.

$\sqcap$ is the binary greatest lower bound, or *meet* operation, and is the dual of the join operation, $\sqcup$, which corresponds to unification, or least upper bounds (in the orientation where $\perp$ corresponds to the most general type). Figure 1 is not a meet semi-lattice because c and d do not have a meet, nor do a and g, for example.

In the finite case, the assumption that every pair of types has a meet is equivalent to the assumption that every consistent set of types, i.e., types with an upper bound, has a join. It is theoretically convenient when discussing the unification of feature structures to assume that the unification of

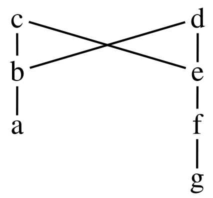

Figure 1: An example of a partial order that is not a meet semi-lattice.

two consistent types always exists. It can also be more efficient to make this assumption as, in some representations of types and feature structures, it avoids a source of non-determinism (selection among minimal but not least upper bounds) during search.

Just because it would be convenient for unification to be well-defined, however, does not mean it would be convenient to think of any empirical domain's concepts as a meet semi-lattice, nor that it would be convenient to add all of the types necessary to a would-be type hierarchy to ensure meet-semi-latticehood. The question then naturally arises as to whether it would be possible, given any finite partial order, to add some extra elements (types, in this case) to make it a meet semi-lattice, and if so, how many extra elements it would take, which also provides a lower bound on the time complexity of the completion.

It is, in fact, possible to embed any finite partial order into a smallest lattice that preserves existing meets and joins by adding extra elements. The resulting construction is the finite restriction of the Dedekind-MacNeille completion (Davey and Priestley, 1990, p. 41).

Definition 2 *Given a partially ordered set,* P, *the Dedekind-MacNeille completion of* P, $\langle DM(P), \subseteq \rangle$, *is given by:*

$$DM(P) = \{A \subseteq P | A^{ul} = A\}$$

This route has been considered before in the context of taxonomical knowledge representation (Aït-Kaći et al., 1989; Fall, 1996). While meet semi-lattice completions are a practical step towards providing a semantics for arbitrary partial orders, they are generally viewed as an impractical preliminary step to performing computations over a partial order. Work on more efficient encoding schemes began with Aït-Kaći et al. (1989), and this seminal paper has

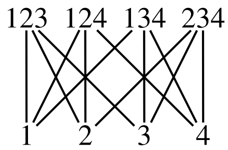

Figure 2: A worst case for the Dedekind-MacNeille completion at $n = 4$.

in turn given rise to several interesting studies of incremental computations of the Dedekind-MacNeille completion in which LUBs are added as they are needed (Habib and Nourine, 1994; Bertet et al., 1997). This was also the choice made in the LKB parsing system for HPSG (Malouf et al., 2000).

There are partial orders P of unbounded size for which $|DM(P)| = \Theta(2^{|P|})$. As one family of worst-case examples, parametrised by n, consider a set $S = \{1, \ldots, n\}$, and a partial order P defined as all of the size $n-1$ subsets of S and all of the size 1 subsets of S, ordered by inclusion. Figure 2 shows the case where $n = 4$. Although the maximum subtype and supertype branching factors in this family increase linearly with size, the partial orders can grow in depth instead in order to contain this.

That yields something roughly of the form shown in Figure 3, which is an example of a recent trend in using type-intensive encodings of linguistic information into typed feature logic in HPSG, beginning with Sag (1997). These explicitly isolate several dimensions[1] of analysis as a means of classifying complex linguistic objects. In Figure 3, specific clausal types are selected from among the possible combinations of CLAUSALITY and HEADEDNESS subtypes. In this setting, the parameter n corresponds roughly to the number of dimensions used, although an exponential explosion is obviously not dependent on reading the type hierarchy according to this convention.

There is a simple algorithm for performing this completion, which assumes the prior existence of a most general element ($\bot$), given in Figure 4. Most instantiations of the heuristic, "where there is no meet, add one" (Fall, 1996), do not yield the Dedekind-MacNeille completion (Bertet et al., 1997), and other authors have proposed incremental methods that trade greater efficiency in computing the entire completion at once for their incrementality.

Proposition 1 *The MSL completion algorithm is correct on finite partially ordered sets, P, i.e., upon termination, it has produced $DM(P)$.*

Proof: Let $V(P)$ be the partially ordered set produced by the algorithm. Clearly, $P \subseteq V(P)$. It suffices to show that (1) $V(P)$ is a complete lattice (with $\top$ added), and (2) for all $v \in V(P)$, there exist subsets $A, B \subseteq P$ such that $v = \bigvee_{V(P)} A = \bigwedge_{V(P)} B$.[2]

Suppose there are $v, w \in V(P)$ such that $v \sqcap w\uparrow$. There is a least element, so v and w have more than one maximal lower bound, l_1, l_2 and others. But then $\{l_1, l_2\}$ is upper-bounded and $l_1 \sqcup l_2\uparrow$, so the algorithm should not have terminated. Suppose instead that $v \sqcup w\uparrow$. Again, the algorithm should not have terminated. So $V(P)$ with $\top$ added is a complete lattice.

Given $v \in V(P)$, if $v \in P$, then choose $A_v = B_v = \{v\}$. Otherwise, the algorithm added v because of a bounded set $\{t_1, t_2\}$, with minimal upper bounds, $u_1, \ldots u_k$, which did not have a least upper bound, i.e., $k > 1$. In this case, choose $A_v = A_{t_1} \cup A_{t_2}$ and $B_v = \bigcup_{1 \leq i \leq k} B_{u_i}$. In either case, clearly $v = \bigvee_{V(P)} A_v = \bigwedge_{V(P)} B_v$ for all $v \in V(P)$. $\Box$

Termination is guaranteed by considering, after every iteration, the number of sets of meet-irreducible elements with no meet, since all completion types added are meet-reducible by definition.

In LinGO (Flickinger et al., 1999), the largest publicly-available LTFS-based grammar, and one which uses such type-intensive encodings, there are 3414 types, the largest supertype branching factor is 19, and although dimensionality is not distinguished in the source code from other types, the largest subtype branching factor is 103. Using supertype branching factor for the most conservative estimate, this still implies a theoretical maxi-

[1]It should be noted that while the common parlance for these sections of the type hierarchy is *dimension*, borrowed from earlier work by Erbach (1994) on multi-dimensional inheritance, these are not dimensions in the sense of Erbach (1994) because not every n-tuple of subtypes from an n-dimensional classification is join-compatible.

[2]These are sometimes called the *join density* and *meet density*, respectively, of P in $V(P)$ (Davey and Priestley, 1990, p. 42).

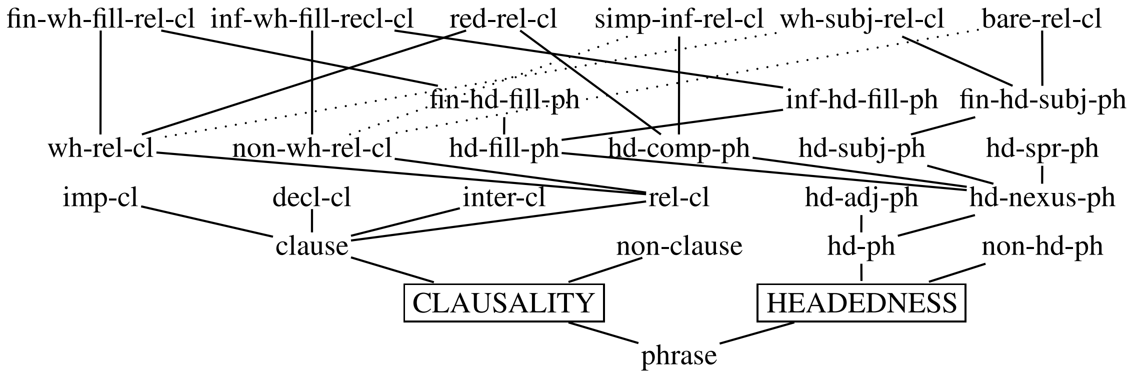

Figure 3: A fragment of an English grammar in which supertype branching distinguishes "dimensions" of classification.

mum of approximately 500,000 completion types, whereas only 893 are necessary, 648 of which are inferred without reference to previously added completion types.

Whereas incremental compilation methods rely on the assumption that the joins of most pairs of types will never be computed in a corpus before the signature changes, this method's efficiency relies on the assumption that most pairs of types are join-incompatible no matter how the signature changes. In LinGO, this is indeed the case: of the 11,655,396 possible pairs, 11,624,866 are join-incompatible, and there are only 3,306 that are consistent (with or without joins) and do not stand in a subtyping or identity relationship. In fact, the cost of completion is often dominated by the cost of transitive closure, which, using a sparse matrix representation, can be completed for LinGO in about 9 seconds on a 450 MHz Pentium II with 1GB memory (Penn, 2000a).

While the continued efficiency of compile-time completion of signatures as they further increase in size can only be verified empirically, what can be said at this stage is that the only reason that signatures like LinGO can be tractably compiled at all is sparseness of consistent types. In other geometric respects, it bears a close enough resemblance to the theoretical worst case to cause concern about scalability. Compilation, if efficient, is to be preferred from the standpoint of static error detection, which incremental methods may elect to skip. In addition, running a new signature plus grammar over a test corpus is a frequent task in large-scale grammar development, and incremental methods, even ones that memoise previous computations, may pay back the savings in compile-time on a large test corpus. It should also be noted that another plausible method is compilation into logical terms or bit vectors, in which some amount of compilation (ranging from linear-time to exponential) is performed with the remaining cost amortised evenly across all run-time unifications, which often results in a savings during grammar development.

3 Unique Feature Introduction

LTFS and ALE also assume that appropriateness guarantees the existence of a unique introducer for every feature:

Definition 3 *Given a type hierarchy,* $\langle T, \sqsubseteq\rangle$*, and a finite set of features, Feat, an* appropriateness specification *is a partial function,* $Approp : Feat \times T \longrightarrow T$ *such that, for every* $\textsc{f} \in Feat$*:*

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

Feature introduction has been argued not to be appropriate for certain empirical domains either, although Pollard and Sag (1994) do otherwise observe it. The debate, however, has focussed on whether to modify some other aspect of type inferencing in order to compensate for the lack of feature introduction, presumably under the assumption that feature introduction was difficult or impossible to restore automatically to grammar signatures that did not have it.

1. Find two elements, t_1, t_2 with minimal upper bounds, $u_1 \ldots u_k$, such that their join $t_1 \sqcup t_2$ is undefined, i.e., $k > 1$. If no such pair exists, then stop.
2. Add an element, v, such that:
   - for all $1 \leq i \leq k$, $v \sqsubseteq u_i$, and
   - for all elements t, $t \sqsubseteq v$ iff for all $1 \leq i \leq k$, $t \sqsubseteq u_i$.
3. Go to (1).

Figure 4: The MSL completion algorithm.

Just as with the condition of meet-semi-latticehood, however, it is possible to take a would-be signature without feature introduction and restore this condition through the addition of extra unique introducing types for certain appropriate features. The algorithm in Figure 5 achieves this. In practice, the same signature completion type, v, can be used for different features, provided that their minimal introducers are the same set, K. This clearly produces a partially ordered set with a unique introducing type for every feature. It may disturb meet-semi-latticehood, however, which means that this completion must precede the meet semi-lattice completion of Section 2. If generalisation has already been computed, the signature completion algorithm runs in $O(fn)$, where f is the number of features, and n is the number of types.

4 Subtype Covering

In HPSG, it is generally assumed that non-maximally-specific types are simply a convenient shorthand for talking about sets of maximally specific types, sometimes called *species*, over which the principles of a grammar are stated. In a view where feature structures represent discretely ordered objects in an empirical model, every feature structure must bear one of these species. In particular, each non-maximally-specific type in a description is equivalent to the disjunction of the maximally specific subtypes that it subsumes.

There are some good reasons not to build this assumption, called "subtype covering," into LTFS or its implementations. Firstly, it is not an appropriate assumption to make for some empirical domains. Even in HPSG, the denotations of

1. Given candidate signature, S, find a feature, F, for which there is no unique introducing type. Let K be the set of minimal types to which F is appropriate, where $|K| > 1$. If there is no such feature, then stop.
2. Add a new type, v, to S, to which F is appropriate, such that:
   - for all $k \in K$, $v \sqsubseteq k$,
   - for all types, t in S, $t \sqsubseteq v$ iff for all $k \in K$, $t \sqsubseteq k$, and
   - $Approp(\text{F}, v) = Approp(\text{F}, k_1) \sqcap Approp(\text{F}, k_2) \sqcap \ldots \sqcap Approp(\text{F}, k_{|K|})$, the generalization of the value restrictions on F of the elements of K.
3. Go to (1).

Figure 5: The introduction completion algorithm.

parametrically-typed lists are more naturally interpreted without it. Secondly, not to make the assumption is more general: where it is appropriate, extra type-antecedent constraints can be added to the grammar signature of the form:

$$n \rightarrow m_1 \vee \cdots \vee m_i$$

for each non-maximally-specific type, n, and its i maximal subtypes, $m_1, \ldots, m_i$. These constraints become crucial in certain cases where the possible permutations of appropriate feature values at a type are not covered by the permutations of those features on its maximally specific subtypes. This is the case for the type, `verb`, in the signature in Figure 6 (given in ALE syntax, where `sub/2` defines the partial order of types, and `intro/2` defines appropriateness on unique introducers of features). The combination, AUX:$-\wedge$ INV:$+$, is not attested by any of `verb`'s subtypes. While there are arguably better ways to represent this information, the extra type-antecedent constraint:

$$\texttt{verb} \rightarrow \texttt{aux\_verb} \vee \texttt{main\_verb}$$

is necessary in order to decide satisfiability correctly under the assumption of subtype covering. We will call types such as `verb` *deranged types*. Types that are not deranged are called *normal types*.

```
bot sub [verb,bool].
  bool sub [+,-].
  verb sub [aux_verb,main_verb]
       intro [aux:bool,inv:bool].
    aux_verb sub [aux:+,inv:bool].
    main_verb sub [aux:-,inv:-].
```

Figure 6: A signature with a deranged type.

4.1 Non-Disjunctive Type Inference under Subtype Covering is NP-Complete

Third, although subtype covering is, in the author's experience, not a source of inefficiency in practical LTFS grammars, when subtype covering is implicitly assumed, determining whether a non-disjunctive description is satisfiable under appropriateness conditions is an NP-complete problem, whereas this is known to be polynomial time without it (and without type-antecedent constraints, of course). This was originally proven by Carpenter and King (1995). The proof, with corrections, is summarised here because it was never published. Consider the translation of a 3SAT formula into a description relative to the signature given in Figure 7. The resulting description is always non-disjunctive, since logical disjunction is encoded in subtyping. Asking whether a formula is satisfiable then reduces to asking whether this description conjoined with `trueform` is satisfiable. Every type is normal except for `truedisj`, for which the combination, DISJ1:`falseform` $\wedge$ DISJ2:`falseform`, is not attested in either of its subtypes. Enforcing subtype covering on this one deranged type is the sole source of intractability for this problem.

4.2 Practical Enforcement of Subtype Covering

Instead of enforcing subtype covering along with type inferencing, an alternative is to suspend constraints on feature structures that encode subtype covering restrictions, and conduct type inferencing in their absence. This restores tractability at the cost of rendering type inferencing sound but not complete. This can be implemented very transparently in systems like ALE that are built on top of another logic programming language with support for constraint logic programming such as SICStus Prolog. In the worst case, an answer to a query to the grammar signature may contain variables with constraints attached to them that must be exhaustively searched over in order to determine their satisfiability, and this is still intractable in the worst case. The advantage of suspending subtype covering constraints is that other principles of grammar and proof procedures such as SLD resolution, parsing or generation can add deterministic information that may result in an early failure or a deterministic set of constraints that can then be applied immediately and efficiently. The variables that correspond to feature structures of a deranged type are precisely those that require these suspended constraints.

```
bot sub [bool,formula].
  bool sub [true,false].
  formula sub [propsymbol,conj,disj,neg,
               trueform,falseform].
    propsymbol sub [truepropsym,
                    falsepropsym].
    conj sub [trueconj,falseconj1,
              falseconj2].
         intro [conj1:formula,
                conj2:formula].
      trueconj intro [conj1:trueform,
                      conj2:trueform].
      falseconj1 intro [conj1:falseform].
      falseconj2 intro [conj2:falseform].
    disj sub [truedisj,falsedisj]
         intro [disj1:formula,
                disj2:formula].
      truedisj sub [truedisj1,truedisj2].
        truedisj1 intro [disj1:trueform].
        truedisj2 intro [disj2:trueform].
      falsedisj intro [disj1:falseform,
                       disj2:falseform].
    neg sub [trueneg,falseneg]
        intro [neg:propsymbol].
      trueneg intro [neg:falsepropsym].
      falseneg intro [neg:truepropsym].
  trueform sub [truepropsym,trueconj,
                truedisj,trueneg].
  falseform sub [falsepropsym,falseconj1,
           falseconj2,falsedisj,falseneg].
```

Figure 7: The signature reducing 3SAT to non-disjunctive type inferencing.

Given a diagnosis of which types in a signature are deranged (discussed in the next section), suspended subtype covering constraints can be implemented for the SICStus Prolog implementation of ALE by adding relational attachments to ALE's type-antecedent universal constraints that will suspend a goal on candidate feature structures with deranged types such as `verb` or `truedisj`. The suspended goal unblocks whenever the deranged type or the type of one of its appropriate features' values is updated to a more specific subtype, and checks the types of the appropriate features' values. Of particular use is the SICStus Constraint Handling Rules (CHR, Frühwirth and Abdennadher (1997)) library, which has the ability not only to suspend, but to suspend until a particular variable is instantiated or even bound to another variable. This is the powerful kind of mechanism required to check these constraints efficiently, i.e., only when necessary. Re-entrancies in a Prolog term encoding of feature structures, such as the one ALE uses (Penn, 1999), may only show up as the binding of two uninstantiated variables, and re-entrancies are often an important case where these constraints need to be checked. The details of this reduction to constraint handling rules are given in Penn (2000b). The relevant complexity-theoretic issue is the detection of deranged types.

4.3 Detecting Deranged Types

The detection of deranged types themselves is also a potential problem. This is something that needs to be detected at compile-time when subtype covering constraints are generated, and as small changes in a partial order of types can have drastic effects on other parts of the signature because of appropriateness, incremental compilation of the grammar signature itself can be extremely difficult. This means that the detection of deranged types must be something that can be performed very quickly, as it will normally be performed repeatedly during development.

A naive algorithm would be, for every type, to expand the product of its features' appropriate value types into the set, A, of all possible maximally specific products, then to do the same for the products on each of the type's i maximally specific subtypes, forming sets B_i, and then to remove the products in the B_i from A. The type is deranged iff any maximally specific products remain in $A \backslash (\cup_i B_i)$. If the maximum number of features appropriate to any type is a, and there are t types in the signature, then the cost of this is dominated by the cost of expanding the products, t^a, since in the worst case all features could have $\perp$ as their appropriate value.

A less naive algorithm would treat normal (non-deranged) subtypes as if they were maximally specific when doing the expansion. This works because the products of appropriate feature values of normal types are, by definition, covered by those of their own maximally specific subtypes. Maximally specific types, furthermore, are always normal and do not need to be checked. Atomic types (types with no appropriate features) are also trivially normal.

It is also possible to avoid doing a great deal of the remaining expansion, simply by counting the number of maximally specific products of types rather than by enumerating them. For example, in Figure 6, `main_verb` has one such product, AUX:$-$ $\wedge$ INV:$-$, and `aux_verb` has two, AUX:$+$ $\wedge$ INV:$+$, and AUX:$+$ $\wedge$ INV:$-$. `verb`, on the other hand, has all four possible combinations, so it is deranged. The resulting algorithm is thus given in Figure 8. Using the smallest normal

For each type, t, in topological order (from maximally specific down to $\perp$):

- if t is maximal or atomic then t is normal. Tabulate normals$(t) = \{t\}$, a minimal normal subtype cover of the maximal subtypes of t.

- Otherwise:

  1. Let $N = \bigcup_{s \in I(t)}$ normals(s), where $I(t)$ is the set of immediate subtypes of t.
  2. Let a be the number of features appropriate to t, and let $R = \{\langle s_1, \ldots, s_a \rangle \mid s_i =$ Approp(F_i, s), Approp$(\mathrm{F}_i, t)\downarrow$, $s \in N\}$.
  3. Given $r_1, r_2 \in R$ such that $r_1 \sqcup r_2 \downarrow$ (coordinate-wise):
     - if $r_1 \sqsubseteq r_2$ (coordinate-wise), then discard r_2,
     - if $r_2 \sqsubseteq r_1$, then discard r_1,
     - otherwise replace $\{r_1, r_2\}$ in R with:

     $\{\langle u_1, \ldots, u_a \rangle \mid u_i$ immed. subtype of s_i in $r_1\}$
     $\cup \{\langle u_1, \ldots, u_a \rangle \mid u_i$ immed. subtype of s_i in $r_2\}$.

     Repeat this step until no such r_1, r_2 exist.
  4. Let $d = \Pi_{\mathrm{F}:\mathrm{Approp}(\mathrm{F},t)\downarrow}$maximal(Approp$(\mathrm{F}, t)) - \Sigma_{\langle u_1, \ldots, u_a \rangle \in R} \Pi_{1 \leq i \leq a}$maximal$(u_i)$, where maximal$(s)$ is the number of maximal subtypes of s.
  5. if $d \neq 0$, then t is deranged; tabulate normals$(t) = N$ and continue. Otherwise, t is normal; tabulate normals$(t) = \{t\}$ and continue.

Figure 8: The deranged type detection algorithm.

subtype cover that we have for the product of t's feature values, we iteratively expand the feature value products for this cover until they partition their maximal feature products, and then count the maximal products using multiplication. A similar trick can be used to calculate *maximal* efficiently.

The complexity of this approach, in practice, is much better: $\mathcal{O}(tb^{da})$, where b is the weighted mean subtype branching factor of a subtype of a value restriction of a non-maximal non-atomic type's feature, and d is the weighted mean length of the longest path from a maximal type to a subtype of a value restriction of a non-maximal non-atomic type's feature. In the Dedekind-MacNeille completion of LinGO's signature, b is 1.9, d is 2.2, and the sum of b^{da} over all non-maximal types with arity a is approximately 10^8. The sum of maximal$^a(t)$ over every non-maximal type, t, on the other hand, is approximately 10^{14}. Practical performance is again much better because this algorithm can exploit the empirical observation that most types in a realistic signature are normal and that most feature value restrictions on subtypes do not vary widely. Using branching factor to move the total number of types to a lower degree term is crucial for large signatures.

5 Conclusion

Efficient compilation of both meet-semi-latticehood and subtype covering depends crucially in practice on sparseness, either of consistency among types, or of deranged types, to the extent it is possible at all. Closure for unique feature introduction runs in linear time in both the number of features and types. Subtype covering results in NP-complete non-disjunctive type inferencing, but the postponement of these constraints using constraint handling rules can often hide that complexity in the presence of other principles of grammar.

References

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

K. Bertet, M. Morvan, and L. Nourine. 1997. Lazy completion of a partial order to the smallest lattice. In *Proceedings of the International KRUSE Symposium: Knowledge Retrieval, Use and Storage for Efficiency*, pages 72–81.

B. Carpenter and P.J. King. 1995. The complexity of closed world reasoning in constraint-based grammar theories. In *Fourth Meeting on the Mathematics of Language*, University of Pennsylvania.

B. Carpenter and G. Penn. 1996. Compiling typed attribute-value logic grammars. In H. Bunt and M. Tomita, editors, *Recent Advances in Parsing Technologies*, pages 145–168. Kluwer.

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

B. A. Davey and H. A. Priestley. 1990. *Introduction to Lattices and Order*. Cambridge University Press.

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

D. Flickinger et al. 1999. The LinGO English resource grammar. Available on-line from `http://hpsg.stanford.edu/hpsg/lingo.html`.

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

T. Frühwirth and S. Abdennadher. 1997. *Constraint-Programmierung*. Springer Verlag.

M. Habib and L. Nourine. 1994. Bit-vector encoding for partially ordered sets. In *Orders, Algorithms, Applications: International Workshop ORDAL '94 Proceedings*, pages 1–12. Springer-Verlag.

R. Malouf, J. Carroll, and A. Copestake. 2000. Efficient feature structure operations without compilation. *Journal of Natural Language Engineering*, 6(1):29–46.

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

G. Penn. 2000b. Applying Constraint Handling Rules to HPSG. In *Proceedings of the First International Conference on Computational Logic (CL2000), Workshop on Rule-Based Constraint Reasoning and Programming, London, UK*.

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

I. A. Sag. 1997. English relative clause constructions. *Journal of Linguistics*, 33(2):431–484.