Flexible Structural Analysis of Near-Meet-Semilattices for Typed Unification-based Grammar Design

Rouzbeh FARAHMAND *Gerald PENN*
University of Toronto
{rouzbeh,gpenn}@cs.toronto.edu

Abstract

We present a new method for directly working with typed unification grammars in which type unification is not well-defined. This is often the case, as large-scale HPSG grammars now usually have type systems for which many pairs do not have least upper bounds. Our method yields a unification algorithm that compiles quickly and yet is nearly as fast during parsing as one that requires least upper bounds. The method also provides a natural naming convention for unification results in cases where no user-defined type exists.

Keywords: HPSG, typed feature structures, unification-based grammar.

Proceedings of COLING 2012: Technical Papers, pages 833–848,
COLING 2012, Mumbai, December 2012.

1 Introduction

In Head-driven Phrase Structure Grammars (HPSG, Pollard and Sag, 1994), most of the on-line computation time is spent performing unifications. HPSG type systems are typically large and involve subtyping, so computing a unification involves computing a *join*, the least informative type that is subsumed by both of the argument feature structures' types.[1] Early TFS-based parsing systems like ALE (Carpenter and Penn, 1996) required that every pair of unifiable types have a least common supertype, i.e., joins are uniquely defined wherever they need to be defined. Type hierarchies that have this requirement are called *meet semi-lattices* (MSLs) (Davey and Priestley, 2002), because the existence of a meet for every pair of types is sufficient to guarantee the existence of a join for every unifiable pair of types when the type hierarchy is finite. Later HPSG parsing systems such as the LKB (Copestake and Flickinger, 2000) and PET (Callmeier, 2000) eliminated this restriction, and it is now extremely rare to find a decent-sized HPSG with an MSL type hierarchy. The LKB, at least in some early versions, used an on-line caching algorithm to add joins to the user-defined type system when necessary. PET adds all of the necessary joins in advance, during a grammar compilation stage. In either case, the time it costs to perform these repairs is proportional to the number of new joins that must be added. It therefore makes some sense to strive to add the least number of types necessary.

While modern HPSG type systems are never MSLs in practice, they are always *almost* MSLs, in a sense that can be made mathematically precise, which we do here for the first time. Converting an HPSG type system to an MSL would be horrendously slow if it were not the case that HPSGs were almost MSLs already, because an exponential number of types need to be added in the worst case. There is another cost hidden in this method, furthermore, at least when we strive to add the least number of extra types necessary. In this case, *naming* these types is also a challenge. Figure 1(a), for example, shows an input type hierarchy that is not an MSL. In Figure 1(b), it has been converted into an MSL by adding one extra type. Ideally, we would like to name this type in a manner that reflects its usage in the grammar. Two possibilities that are immediately evident are $e \vee f$ or $a \wedge b \wedge c \wedge d$. Neither of these are ideal because they do not attest to a precise pair of types that the system needed to unify during parsing, and which therefore gave rise to the need for this extra type. This is extremely useful for debugging understanding feature structure outputs with large grammars. Grammar writers instantly know what user-defined types had been combined.

The key to the alternative introduced here is that it actually costs *less* time in practice to compute with a *larger* type system provided that joins are added on-line when necessary. This is achieved by doing a small amount of extra compilation off-line — small in practice again because of a specific mathematical property that practical HPSGs possess - and using the data structure that it creates to add many more types during parsing than the absolute minimum necessary for MSL-hood. In the limit, our method creates a type hierarchy more like a *conjunctive lattice,* but it only adds types as it needs them. Figure 1(c) is what it creates from Figure 1(a). From the standpoint of naming conventions, this type system is ideal, because every added type is named for the set (pair or larger) of types that required it.

[1] Some of the HPSG community draw these subtypes below their supertypes in graphical representations of type hierarchies, but still use the name *join*, even though the correct term is *meet* in this orientation. We will also use the term, *join*, but will depict type hierarchies in the opposite orientation, in which more specific subtypes appear above their more general supertypes.

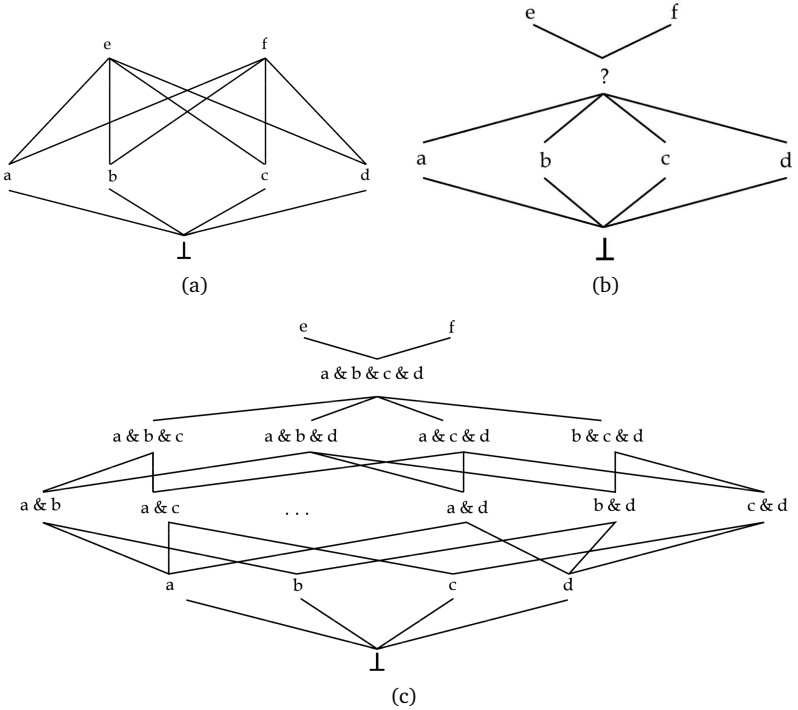

Figure 1: (a) A non-MSL type hierarchy, (b) its MSL completion based on the Dedekind-MacNeille completion, and (c) its conjunctive lattice.

Section 2 discusses the mathematical background necessary to understand the conversion that yields Figure 1(b). Section 3 presents a precise way of appreciating what we mean by a "near-meet-semilattice," through quantifying the size and number of a special kind of subset of types called *prime sets*. Section 4 then introduces another special kind of subset that can be used to efficiently compute the prime sets of a type hierarchy. Section 4.1 describes how joins are computed with the new method, and what the compiler computes to achieve this. Section 6 then presents an evaluation which shows that the new method has nearly the same performance at run-time (5.4% slower) as parsing with precompiled MSLs directly, but without the compile-time cost of the latter. We assert that in the context of grammar design, 5% is worth less compilation time along with the better naming conventions that naturally ensue from this method. Irrespective of compiling and parsing speed, the formal treatment of this subject also provides a more refined theory for analyzing the structure of type hierarchies in large-scale grammars than a binary MSL/non-MSL distinction.

2 Dedekind-MacNeille Completions

Let P be a partially ordered set, and $S \subseteq P$. The set of upper bounds of S, written S^u, is the set of all $x \in P$ such that for all $s \in S, s \leq_P x$. The lower bounds, S^l, are all of the $x \in P$ such that for all $s \in S, x \leq_P s$. The *Dedekind-MacNeille completion* of P ($DM(P)$) is the set of all $A \subseteq P$ such that $A^{ul} = A$. All of the singleton subsets of P are included in $DM(P)$, so informally we can say that P is included in $DM(P)$. $DM(P)$ is the smallest set that both includes P and is an MSL (Davey and Priestley, 2002).

The computer science literature on lattice theory has tended to emphasize $DM(P)$ as either a theoretical device for understanding (but not computing) joins even when they are not present (e.g., Aït-Kaci et al., 1989), or a goal to aim for when adding joins incrementally, each as it becomes necessary (e.g., Bertet et al., 1997). Using $DM(P)$ is supposed to provide us with some sort of reassurance about the maximum amount of extra work that must be performed to embed P into an MSL, at least if the results are pre-compiled or cached. Because $|DM(P)|$ is exponential in $|P|$ in the worst case, however, this does not provide much solace by itself. What matters in the end is the actual amount of overhead accrued for working with a non-MSL type hierarchy. This is true even if $DM(P)$ is not used.

That overhead can be paid either all at once during an initial compilation stage, or in small amounts over time, with the hope that some completion types will never be needed by the user's queries. Obviously, we do not want to compute any completion larger than $DM(P)$ at compile-time if we can help it. The prospect of incrementally computing completion types in a larger lattice is still available, however, as long as the overhead is still acceptable in practice. It is also worth noting that the incremental algorithm of Bertet et al. (1997) includes a line which stops execution for every type that is added and prompts the user to name it. The problem of naming in the Dedekind-MacNeille completion extends well beyond applications to computational linguistics.

3 Prime Sets

What remains then is to find a new source of reassurance about on-line cost that depends upon structural properties of the input type hierarchy rather than upon the size of the worst-case lattice using a given completion strategy. The best structural property of all is for P to be an MSL already. In this case, $DM(P)$ consists only of singleton sets plus possibly a new top-most element (which is usually discarded anyway), and so $|DM(P)| \leq |P| + 1$.

Paradoxically, even if P were not an MSL, finding the upper bound of a set of arguments $S \subseteq P$ in many cases is as easy as combining every type of S in succession, according to some arbitrary linear ordering of its elements. At each step, we compute the join of the current element and a running accumulator type, or fail if they have no upper bounds in common. That is still linear time in $|S|$, even if non-linear in other variables, as is the case when P is an MSL. It is difficult to imagine doing any better in this particular dimension.

The trouble is that this linear-time incremental method does not always work, because of subsets of S (or P) that we will call *prime sets*.[2].

Definition 1. *Let P be a partially ordered set, and $S \subseteq P$. S is an anti-chain iff for all $x, y \in S$, neither $x \leq_P y$ nor $y \leq_P x$.*

[2]These should not be confused with prime ideals or filters nor the prime elements that generate them (see section I-3 of Gierz et al., 2003)

Definition 2. *Let P be a partially ordered set, and $S \subseteq P$. S is a prime set of P iff $|S| > 1$, S is an anti-chain, and, for all non-empty $T \subseteq S$, T has a join iff $|T| = |S|$ or $|T| = 1$.*
Proposition 3. *A partial order is a meet-semi-lattice iff its maximal prime sets are of size 2.*

No non-trivial proper subset of a prime set has a join. When $|S| > 2$ and S is prime, considering pairs of elements in succession will not work for any linear ordering of S. In Figure 2(a), for example, $S = \{a, b, c\}$ is a prime set because it has a join and none of its 2-subsets does. One needs to consider all three at once to see the join.

Candidate sets S of size larger than 2 are important because they naturally result from delaying the computation of joins. A very natural strategy for performing unification in a non-MSL is to use joins where they exist, and the union of the sets of types to be unified otherwise, hoping that the latter will eventually be resolved to a join by the addition of some other element later in the computation. This strategy implicitly uses the conjunctive lattice of P to support its computations, but does not compute it. It is also the strategy that we will adopt. The only thing we need to remember is that we should never refer to a set or subset of types when that set has a join in P that we could refer to instead. That amounts to using the user's names for conjunctions vis-a-vis the subtyping relation where they exist, and explicit conjunctions elsewhere. Operationally, it reduces unification to searching for prime sets and prime subsets.

In a nutshell, the reason that $DM(P)$ can be exponentially larger than P is that the prime sets of P can grow in size linearly with $|P|$, as will be proven in the next section. Better still, tabulating the number of prime sets of each size indicates how "MSL-like" a type hierarchy is. It gives us a spectral view of the basic subsets that every large set decomposes into during unification.

While most large HPSGs are not MSLs in practice, they do not avail themselves of the possibility of having large prime sets either. This is evident in their spectra, and it means that it is relatively inexpensive to enumerate all of the prime sets.

4 Pseudo-prime Sets

A close variant of prime sets turns out to be even more useful:
Definition 4. *$S \subseteq P$ is a pseudo-prime set of P iff $|S| > 1$, S is consistent, and, for all non-empty $T \subseteq S$, T has a join iff $|T| = 1$.*

A pseudo-prime set is as close as a set with no least upper bound can get to being a prime set. The number of pseudo-prime sets is also an upper bound on the potential number of new types added by a completion, because every completion type in $DM(P)$ corresponds to the set of upper bounds of some pseudo-prime set (Penn, 2000, though stated in different terms).

In Figure 2(a), $\{a, b\}, \{a, c\}$ and $\{b, c\}$ are all pseudo-prime sets, for example.

Pseudo-prime sets stand in a very special relationship to prime sets and to each other:
Proposition 5. *Every k-subset of a pseudo-prime set with $k > 1$ is a pseudo-prime set.*
Proposition 6. *Every proper k-subset of a prime set with $k > 1$ is a pseudo-prime set.*

This means that we do not have to search through all $\binom{|P|}{k}$ possible k-subsets of P to find the size-k prime sets of P. Instead, we can merely focus on the size $k-1$ pseudo-primes,

```
Data: A finite poset of types P = ⟨T,⊑⟩
Result: All the pseudo-prime PsPrime(P) and prime sets Prime(P) of P
1  begin
2      init: Prime(P) ← {}, PsPrime(P) ← {} ;
3      forall the consistent types t1,t2 ∈ T do
4          Mins ← FindMinimal({t1,t2}^u);
5          if Mins has only one element then                    // There is a least upper bound
6              Add {t1,t2} to Prime(P) ;
7          else// There is no join
8              Add {t1,t2} to PsPrime(P) ;
9          end
10     end
11     if if there is no set in PsPrime(P) with size 2 then
12         return PsPrime(P) and Prime(P) ;
13     else// P is not an MSL: search for primes and pseudo-primes of size greater than 2
14         size := 2 ;
15         repeat
16             forall the pp1,pp2 ∈ PsPrime(P) do
17                 Cand ← pp1 ∪ pp2;
18                 if |Cand| is size + 1 then
19                     Mins ← FindMinimal(Cand);
20                     if Mins has only one element then
21                         Prime(P) += Cand;
22                     else
23                         PsPrime(P) += Cand;
24                     end
25                 end
26             end
27             size = size + 1;
28         until (there is no pseudo-prime set of size);
29         return PsPrime(P) and Prime(P) ;
30     end
31 end
```

Algorithm 1: Algorithm for finding prime and pseudo-prime sets of a poset.

one of which must live inside every prime set. The size $k-1$ pseudo-primes in turn can be constructed from $k-2$-pseudo-primes, etc., with the 2-pseudo-primes being the pairs of types that attest to the non-MSLhood of P. This recursive search procedure is given in Algorithm 1. Any procedure that enumerates prime sets or pseudo-prime sets is exponential in the worst case, but this algorithm makes efficient use of the relationship between primes and pseudo-primes to achieve efficiency when the largest prime sets are of low cardinality.

Pseudo-primes are also the key to bounding the size of prime sets:

Proposition 7. *The maximum size attainable by a prime subset of a poset P is $\lfloor \frac{|P|-1}{2} \rfloor$.*

Proof. Let Ψ be any prime subset of P. Let $n = |P|$ and $p = |\Psi|$. Consider the p distinct subsets of Ψ of size $p-1$ (i.e. $K_1, ..., K_p \subseteq \Psi$ such that $|K_1| = ... = |K_p| = p-1$). Let $m = |\bigcup_{1 \le i \le p} \mu(K_i)|$ where $\mu(S) = \{x \in S^u \mid \nexists\, u \in S^u \text{ s.t } u \le x\}$, the set of all minimal upper-bounds of S. For all i, $|\mu(K_i)| \ge 2$ because $K_1, ..., K_p$ are all pseudo-prime sets. For the same reason, $|\mu(K_i) \cap \mu(K_j)| \le 1$ when $i \ne j$, because then $K_i \cup K_j = \Psi$. In fact, $\mu(K_1), ..., \mu(K_p)$ are either all pairwise disjoint or all agree on the same one element, so there are only two cases:

Case A: $\bigcap_i \mu(K_i) = \emptyset$ Then P must have at least $p+m$ elements and one extra element for the join of Ψ that must exist above all the m elements in $\bigcup_{1 \le i \le p} \mu(K_i)$. So, (i) $n \ge p+m+1$. Conservatively, we must choose p disjoint antichains of size at least 2 from the m elements, thus we have (ii) $p \le \lfloor \frac{m}{2} \rfloor$. Therefore, from (i) and (ii), if m is even then $n \ge 2p+1$, and if m is odd then $n \ge 3p+2$. Since $2p+1 \le 3p+2 \le n$, then $p \le \frac{n-1}{2}$.

Case B: $\bigcap_i \mu(K_i) \neq \emptyset$. In this case, $n \geq p + m$ and $p \leq m - 1$, subtracting for the one upper bound that all of the K_i have in common. Therefore, $n \geq p + m \geq 2p + 1$ and consequently $p \leq \frac{n-1}{2}$.

□

There are also simple examples in which this bound can be attained, so this is tight.

A spectral decomposition of pseudo-primes is also somewhat interesting because of its relationship to the size of $DM(P)$, and to prime sets — the largest prime set cannot be more than one element larger than the largest pseudo-prime, but in practice it is much less. Figure 3 shows the decompositions for both primes and pseudo-primes for both a 1999 pre-release of the English Resource Grammar (ERG Flickinger, 1999) and Berligram (Müller, 2007). The first thing that we can observe is the very small size of all of the pseudo-primes and primes in both grammars — at most 9 in the ERG, and 6 in Berligram. It took 27 seconds [3] to find all the primes and pseudo-primes in the ERG, and less than 1 second to find them in Berligram. By contrast, it takes the LKB 4 seconds to compute $DM(P)$ for the ERG. We can also see a greater difference between the largest prime and the largest pseudo-prime in the ERG (9-4=5) than we can in Berligram (6-4=2). Pseudo-primes lay the groundwork for primes, so when a prime set nevertheless does not occur, this is significant. Finally, we can observe that, although the ERG is nearly 10 times the size of Berligram in its total number of types, it has about 25 times as many primes of cardinality 3 or greater, and of the same maximum size (4).

A large number of primes, a skewed distribution of primes towards small cardinalities, a large number of pseudo-primes, and a large difference between the size and/or number of pseudo-primes and primes are all strong indicators that a type hierarchy is more MSL-like when interpreted relative to the total number of types. The total number of types is not a good indicator, nor is "join density," which is common to formal concept analysis (Besson et al., 2005).[4] The (relative to P) size of $DM(P)$ is not bad, but it fails to indicate just how far the tails of these distributions extend. A type hierarchy with 10^6 pseudo-primes of size 2 is more MSL-like than one with 10^5 pseudo-primes of size 9, and simple counts of $|DM(P)|$ often cannot tell the difference, especially with high join densities.

One conclusion to draw from this particular comparison is obviously that the ERG is more MSL-like, which implies that it is more conservative in its structure, and would be easier to compute with than many other potential 3414-type hierarchies. On the other hand, the sheer number of types may make finding the right part of the type hierarchy to modify more difficult than it needed to have been. Another way to look at it is that Berligram makes more efficient use of the amount of information that a 434-type hierarchy can carry. There is more information embedded in its structure relative to its size, but that information may make it more difficult to predict the consequences of type unification than in some other hierarchies of its size. The ERG and Berligram do not have the same number of types, nor

[3] All the timing results reported in this paper were obtained on an Intel®-based system with a 3.6 GHz Xeon™ processor and 3 GB of RAM running the Ubuntu 6.06.2 Linux operating system.

[4] Join density, in terms of our partial orders, is calculated as the ratio of the size of the extension of the (transitively and reflexively closed) subtype relation to the square of $|P|$. The join density of the ERG is .001; that of Berligram is .006.

do they attempt to account for the same constructions, nor even the same language, so we can only speculate here. But prime sets and pseudo-primes can illuminate these issues.

4.1 Building an Automaton-based Index

The final application of prime and pseudo-prime sets considered here is to unification itself. As mentioned above, it is possible to maintain argument sets of types from a non-MSL as argument sets even after unification, provided that all of the prime subsets are replaced with their joins. For the unifier not to perform this replacement is tantamount to it simply handing back a candidate pair of types from an MSL ununified.

If we use topologically sorted lists (as induced by the subtype relation) as our representations of prime sets, pseudo-prime sets and argument sets, then Propositions 5 and 6 will allow us to use a simple automaton-based index to perform this replacement. The index has a linear number of states relative to the number and size of the pseudo-primes, because each state corresponds to a prefix of one or more pseudo-primes in topological order. It has two kinds of edges: *suffix edges* and *redex edges*, and both kinds of edges are labelled with elements of pseudo-prime or prime sets.

All of the pseudo-prime sets can be arranged into a tree such that every path through the tree from the root to any node corresponds to a pseudo-prime or a singleton set. Paths from the root to a leaf correspond to maximal pseudo-primes — no pseudo-prime contains them. These trees are connected with suffix edges, each of which is labelled with the k^{th} element that extends a $k-1$-length prefix of one or more pseudo-primes to a k-length prefix. On top of this skeleton, we add the redex edges, which map a pseudo-prime prefix, X, that contains a $k-1$-length prefix of a size-k prime set to the result of replacing that prime set within X by its join, transitively closed under all possible further replacements. The redex edge is labelled with the k^{th} element that completes the prime set.

The index for Figure 2(a), for example, is shown in Figure 2(b). Its pseudo-prime sets are: $\{\{a,b\}_{12}, \{a,c\}_{13}, \{a,z\}_{14}, \{b,z\}_{15}, \{b,c\}_{16}, \{d,z\}_{17}, \{c,z\}_{18}, \{g,z\}_{19}, \{f,z\}_{20}, \{e,z\}_{21}, \{h,z\}_{22}, \{a,b,z\}_{23}, \{a,c,z\}_{24}, \{b,c,z\}_{25}$; and its prime sets are: $\{a,f\}$, $\{b,g\}$, $\{d,c\}$, $\{d,g\}$, $\{d,f\}$, $\{d,e\}$, $\{g,f\}$, $\{g,e\}$, $\{f,e\}$, $\{a,b,c\}$.[5] Suppose $\{a,b,c,z\}$ is of interest. The automaton consumes a and b, which is a pseudo-prime (12) and the prefix of the pseudo-prime, $\{a,b,z\}_{23}$, but upon seeing c traverses a redex edge that reduces the prime set $\{a,b,c\}$ to its join, e, which is a singleton and therefore not subject to further replacement. Consuming z next leads to the pseudo-prime, $\{e,z\}_{21}$. So if the union of the argument sets provided to the unifier is $\{a,b,c,z\}$, what the unifier should return is $\{e,z\}$, which should be interpreted as the conjunctive type, $e\&z$. This is as close to the grammar writer's idioms as we can come in describing this result, given that the type hierarchy is not an MSL.

Having constructed the automaton directly from the pseudo-prime and prime sets, it can then be pruned by eliminating states that correspond to singletons with no outgoing suffix edges and no outgoing or incoming redex edges. Singleton sets clearly do not need to be fed to the automaton. In the example in Figure 2(b), states 9, 10 and 11 would be pruned.

Some statistics for the automata constructed for the ERG and Berligram are presented in

[5] These sets are sorted according to the following topological order (subscripts are topological ordinals): $\bot_0 < a_1 < b_2 < d_3 < c_4 < g_5 < f_6 < e_7 < h_8 < z_9 < y_{10} < x_{11}$.

Table 1. Our implementation of the automaton compiler has been written in Prolog and

	Before pruning		After pruning	
	ERG	Berligram	ERG	Berligram
# of suffix edges	9508	1425	7058	1205
# of redex edges	2296	554	2296	554
Total # of edges	11804	1979	9354	1759
Total # of states	8967	1314	6517	1094

Table 1: Statistics related to the constructed automaton-based index for type hierarchies of the ERG (Flickinger, 1999) and Berligram (Müller, 2007).

generates the automaton as Prolog clauses, too, which are then also compiled. The two stages of compilation together consumed 30 milliseconds for Berligram and 2.2 seconds for the ERG, which is well within the range of compilers that add joins on-line. Further speed improvements are doubtlessly possible, since the automaton can in principle be constructed incrementally as Algorithm 1 is being executed.

5 Unification

Having constructed the automaton directly from pseudo-prime and prime sets, we are implicitly assuming that every input set to the automaton is an anti-chain. The algorithm for unification, given as Algorithm 2, shows how the result should look.

Data: $P = \langle T, \sqsubseteq_P \rangle$;
two sets of types $A, B \subseteq T$;
the automaton-based index $\mathscr{M}_P$;
ψ: the topology of P used for construction of $\mathscr{M}_P$;
Result: minimal upper bound of A and B or failure

```
1 begin
2     Feed ← AntiChainReduce(P, ψ ,A∪B);
3     if Feed has only one element then
4         return Feed;
5     else
6         AutResult ← Aut(start state of M_P , Feed);
7         return AutResult;
8     end
9 end
```

Algorithm 2: The algorithm for unification of two sets of types.

`AntiChainReduce` reduces an input argument set to the topologically sorted, maximally specific anti-chain that contains it. This can be accomplished in quadratic time as a function of the size of the input set, simply by considering every pair of types and replacing the pair with the higher of the two if they are ordered.

6 Parsing Evaluation

To evaluate how this unification algorithm performs in practice, we evaluated three systems on a common grammar and corpus of sentences. The grammar was the ERG, and the corpus, called the FUSE corpus, has 2354 sentences, ranging from 1 to 50 words in length, 192 of which are ungrammatical according to the ERG. All three of the systems that we

tested had memory allocation problems while parsing this corpus, so every sentence that resulted in at least one system running out of memory was excluded, leaving a remainder of 1727 sentences, 159 of which were ungrammatical.[6] We also imposed a maximum of 8000 chart edges after which parsing was terminated. The excluded sentences were those that exceeded the memory available to one of the processes with that cap in place. The three systems were:

(1) ALE version 4.0 beta, running on the Dedekind-MacNeille completion of an ALE port of a 1999 pre-release version of the ERG, with 45 rules, 155 features, 1314 lexical entries, no lexical rules and 3412 types, plus a most general type, $\bot$, plus a built-in type for strings. $DM(P)$ added another 893 types to the system. We compiled ALE with SICStus Prolog 3.12.10 (compact code). [7]

(2) the LKB, version as at March, 2009: This system allows non-MSL grammars as input, but it automatically computes the Dedekind-MacNeille completion on their type systems at compile-time, naming newly added types with a number. The 1999 pre-release of the ERG no longer runs on the LKB because of non-backwards-compatible changes in the system over the last 10 years. Porting an LKB grammar to ALE is very tricky because the LKB's input syntax is very heavily overloaded. So instead of porting the current ERG to ALE, we reported our existing port of the 1999 pre-release of the ERG back to this version of the LKB. We have verified that these two ports generate exactly the same edges on the non-excluded sentences of this corpus. We compiled the LKB with Allegro Common Lisp Enterprise 8.0.

(3) an experimental system, obtained by replacing ALE 4.0's type unifier with the one described in this paper. This system generates the same object code as ALE 4.0 when the input grammar's type system is an MSL. We ran this on the 1999 pre-release of the ERG without computing the Dedekind-MacNeille completion, which thus results in different object code.

The results are shown in Figure 6. These are log-linear graphs, so the roughly even separations attest to a nearly constant factor of speed-up, not a constant difference in milliseconds. The LKB is approximately 1.82 times slower than ALE 4.0, and the experimental version implemented for this study is approximately 1.054 times slower than ALE 4.0. Thus the experimental system produces a run-time parser that is 5.4% slower than ALE after having compiled out the Dedekind-MacNeille completion in advance, but is still more than 40% faster than the LKB, a commonly used system that caches them on-line.

The figure of 5.4% obtains with an experimental system that does not cache the results of prime-set reduction on previously seen argument sets. Caching is best implemented by caching not only previously seen argument sets that are consistent, but also previously seen argument sets that are inconsistent. As shown in Figure 6(d), however, the difference is not large. In the latter case, one obtains a system that is an average of 5.7% slower than parsing with precompiled MSLs on a first pass of parsing, but only 2.7% slower on a second pass through the same corpus of sentences (for which all of the cache entries are in place). We used the built-in SICStus `term_hash/2` predicate to hash argument sets, which gave us a perfect hash (one set per hash). Here again, a comparison with the LKB is informative in that argument set caching behaves surprisingly like the LKB's lexicon caching: while we

[6] The test sentences and results are available at `http://www.cs.toronto.edu/~rouzbeh/resources.html`.
[7] The MSL-ERG in ALE syntax is available at `http://www.cs.toronto.edu/~rouzbeh/resources.html`.

might naturally expect to see a steady stream of new words throughout the parsing of a corpus of this size, we also see a steady stream of new argument sets, even though the number of types and their relative ordering are held constant throughout the experimental parsing runs. As a result, it would take a much larger sample to witness a convergence of the second pass with the first. This suggests that argument set caching is only worthwhile over much longer timespans of use with the same type system.

7 Conclusion

We introduced two new mathematical constructions, prime sets and pseudo-prime sets, and showed that they provide a reasonable alternative to the Dedekind-MacNeille completion, by providing a means for manipulating conjunctive sets of types at very low overhead. These sets provide better naming conventions for newly added types than any implementation based on Dedekind-MacNeille could hope to do because they effectively allow for a trace of partial unifications of a set of arguments. Prime and pseudo-prime sets also form the basis of a more refined method for analyzing the upper bounds of a type system than simply calling it an MSL or non-MSL. This is of independent value to grammar designers.

We have not yet looked at the frequency distributions of primes and pseudo-primes encountered during parsing with the ERG over the FUSE corpus, for example. Both our compilation strategy and the spectral decomposition method would benefit from taking this information into account, because some portions of a completed lattice are clearly more important than others, simply as a result of certain types being used more often in parsing representative corpus input than others.

Some additional work on the combinatorial properties of prime and pseudo-prime sets also remains. Although we have identified a tight upper bound on the possible sizes of prime sets, the only known bounds on the numeracy of prime and pseudo-prime sets are based on the classical problem (the so-called *Dedekind Problem*), of finding the number of anti-chains of a given poset. The field had settled on a fairly stable bound until recently (Korshunov and Shmulevich, 2000).

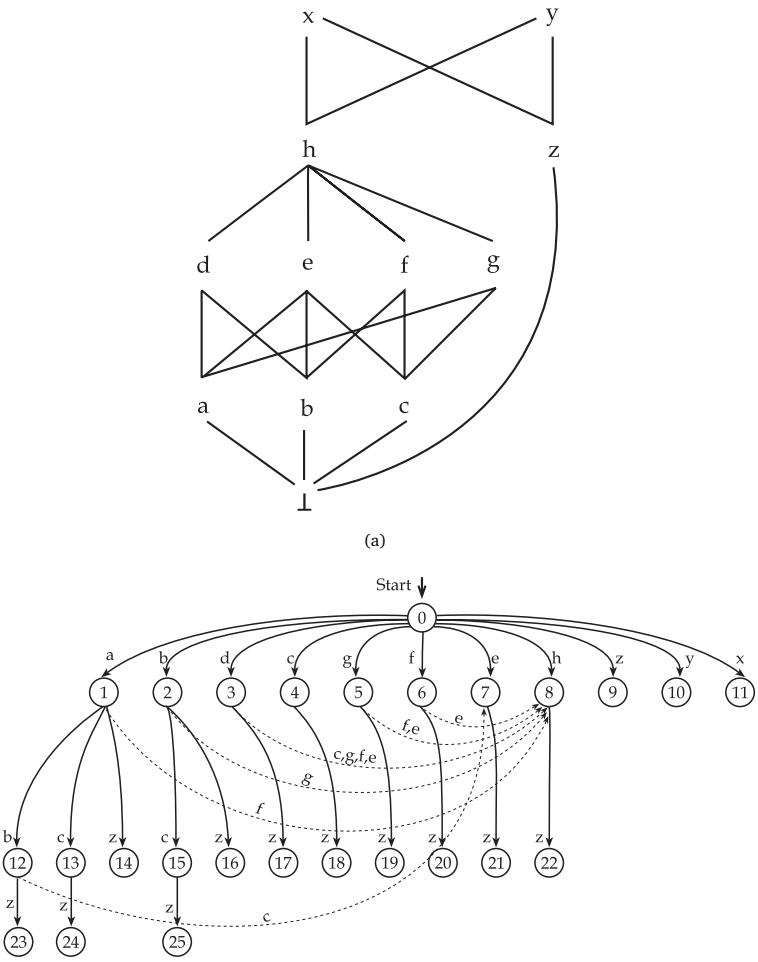

Figure 2: (a) A non-MSL type hierarchy and (b) its automaton-based index). The redex edges are depicted as dotted arcs; if those eliminated from the automaton, a suffix tree is obtained.

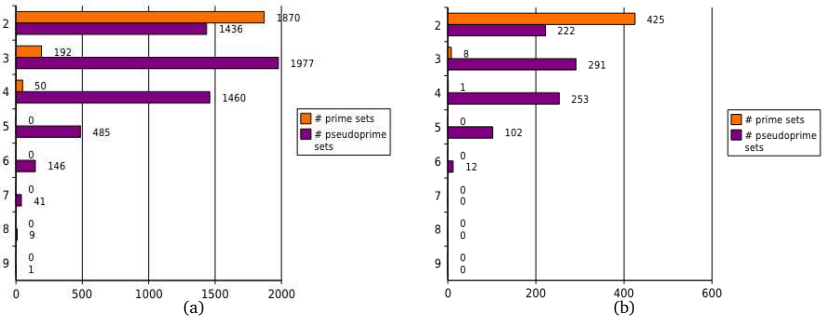

Figure 3: Number and size of the prime and pseudo-prime sets of (a) the English Resource Grammar (3412 types in total), and (b) the German Berligram (434 types). The horizontal and vertical axes show the number and size of the sets, respectively.

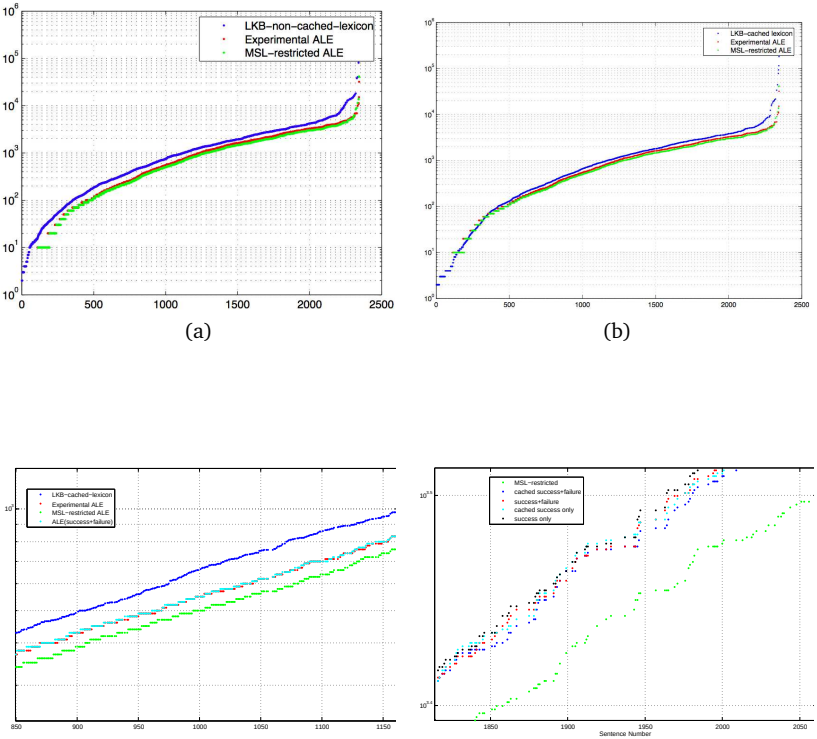

Figure 4: Evaluation of the `MSL-restricted ALE`, `Experimental ALE` and the LKB on FUSE. The LKB caches the lexicon as it parses, so each sentence was parsed twice in succession. The parsing time of the first parse is given in (a) and that of the second in (b). Figure (c) is a close-up of a portion of Figure (b), along with Experimental ALE plus caching of consistent and inconsistent argument sets (first pass). Figure (d) shows the effect of caching in close-up detail.

References

Aït-Kaci, H., Boyer, R., Lincoln, P., and Nasr, R. (1989). Efficient implementation of lattice operations. *ACM Trans. Program. Lang. Syst.*, 11(1):115–146.

Bertet, K., Morvan, M., and Nourine, L. (1997). Lazy completion of a partial order to the smallest lattice.

Besson, J., Robardet, C., Boulicaut, J.-F., and Rome, S. (2005). Constraint-based concept mining and its application to microarray data analysis. *Intell. Data Anal.*, 9(1):59–82.

Callmeier, U. (2000). Pet – a platform for experimentation with efficient HPSG processing techniques. *Nat. Lang. Eng.*, 6(1):99–107.

Carpenter, B. and Penn, G. (1996). Efficient parsing of compiled typed attribute value logic grammars. In Bunt, H. and Tomita, M., editors, *Recent Advances in Parsing Technology*, pages 145–168. Kluwer Academic Publishers, Dordrecht, Boston, London.

Copestake, A. and Flickinger, D. (2000). An open-source grammar development environment and broad-coverage English grammar using HPSG. In *In Proceedings of LREC 2000*.

Davey, B. and Priestley, H. (2002). *Introduction to Lattices and Order*. Cambridge University Press.

Flickinger, D. (1999). LinGo, the English Resource Grammar.

Gierz, G., Hofmann, K. H., Keimel, K., Lawson, J. D., Mislove, M., and Scott, D. S. (2003). *Continuous Lattices and Domains*, volume 93 of *Encyclopedia of Mathematics and its Applications*. Cambridge.

Korshunov, A. D. and Shmulevich, I. (2000). On the distribution of the number of monotone boolean functions relative to the number of lower units. *Discrete Math.*, 257(2-3):463–479.

Müller, S. (2007). Berligram: German grammar based on Head-driven Phrase Structure Grammar: Eine Einführung.

Penn, G. (2000). *The Algebraic Structure of Attributed Type Signatures*. PhD thesis, School of Computer Science, Carnegie Mellon University.

Pollard, C. and Sag, I. (1994). *Head-Driven Phrase Structure Grammar*. Chicago University Press, Chicago, Illinois.