A Generalized-Zero-Preserving Method for Compact Encoding of Concept Lattices

Matthew Skala
School of Computer Science
University of Waterloo
mskala@cs.toronto.edu

Victoria Krakovna
János Kramár
Dept. of Mathematics
University of Toronto
{vkrakovna,jkramar}@gmail.com

Gerald Penn
Dept. of Computer Science
University of Toronto
gpenn@cs.toronto.edu

Abstract

Constructing an encoding of a concept lattice using short bit vectors allows for efficient computation of join operations on the lattice. Join is the central operation any unification-based parser must support. We extend the traditional bit vector encoding, which represents join failure using the zero vector, to count any vector with less than a fixed number of one bits as failure. This allows non-joinable elements to share bits, resulting in a smaller vector size. A constraint solver is used to construct the encoding, and a variety of techniques are employed to find near-optimal solutions and handle timeouts. An evaluation is provided comparing the extended representation of failure with traditional bit vector techniques.

1 Introduction

The use of bit vectors is almost as old as HPSG parsing itself. Since they were first suggested in the programming languages literature (Aït-Kaci et al., 1989) as a method for computing the unification of two types without table lookup, bit vectors have been attractive because of three speed advantages:

- The classical bit vector encoding uses bitwise AND to calculate type unification. This is hard to beat.
- Hash tables, the most common alternative, involve computing the Dedekind-MacNeille completion (DMC) at compile time if the input type hierarchy is not a bounded-complete partial order. That is exponential time in the worst case; most bit vector methods avoid explicitly computing it.
- With large type signatures, the table that indexes unifiable pairs of types may be so large that it pushes working parsing memory into swap. This loss of *locality of reference* costs time.

Why isn't everyone using bit vectors? For the most part, the reason is their size. The classical encoding given by Aït-Kaci et al. (1989) is at least as large as the number of meet-irreducible types, which in the parlance of HPSG type signatures is the number of unary-branching types plus the number of maximally specific types. For the English Resource Grammar (ERG) (Copestake and Flickinger, 2000), these are 314 and 2474 respectively. While some systems use them nonetheless (PET (Callmeier, 2000) does, as a very notable exception), it is clear that the size of these codes is a source of concern.

Again, it has been so since the very beginning: Aït-Kaci et al. (1989) devoted several pages to a discussion of how to "modularize" type codes, which typically achieves a smaller code in exchange for a larger-time operation than bitwise AND as the implementation of type unification. However, in this and later work on the subject (e.g. (Fall, 1996)), one constant has been that we know our unification has failed when the implementation returns the zero vector. *Zero preservation* (Mellish, 1991; Mellish, 1992), i.e., detecting a type unification failure, is just as important as obtaining the right answer quickly when it succeeds.

The approach of the present paper borrows from recent statistical machine translation research, which addresses the problem of efficiently representing large-scale language models using a mathematical construction called a *Bloom filter* (Talbot and Osborne, 2007). The approach is best combined with modularization in order to further reduce the size of the codes, but its novelty lies in

Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics, pages 1512–1521,
Uppsala, Sweden, 11-16 July 2010.

the observation that counting the number of one bits in an integer is implemented in the basic instruction sets of many CPUs. The question then arises whether smaller codes would be obtained by relaxing zero preservation so that *any* resulting vector with at most λ bits is interpreted as failure, with $\lambda \geq 1$.

Penn (2002) generalized join-preserving encodings of partial orders to the case where more than one code can be used to represent the same object, but the focus there was on codes arising from successful unifications; there was still only one representative for failure. To our knowledge, the present paper is the first generalization of zero preservation in CL or any other application domain of partial order encodings.

We note at the outset that we are not using Bloom filters as such, but rather a derandomized encoding scheme that shares with Bloom filters the essential insight that λ can be greater than zero without adverse consequences for the required algebraic properties of the encoding. Deterministic variants of Bloom filters may in turn prove to be of some value in language modelling.

1.1 Notation and definitions

A partial order $\langle X, \sqsubseteq \rangle$ consists of a set X and a reflexive, antisymmetric, and transitive binary relation $\sqsubseteq$. We use $u \sqcup v$ to denote the unique least upper bound or *join* of $u, v \in X$, if one exists, and $u \sqcap v$ for the greatest lower bound or *meet*. If we need a second partial order, we use $\preceq$ for its order relation and $\curlyvee$ for its join operation. We are especially interested in a class of partial orders called *meet semilattices*, in which every pair of elements has a unique meet. In a meet semilattice, the join of two elements is unique when it exists at all, and there is a unique globally least element $\bot$ ("*bottom*").

A *successor* of an element $u \in X$ is an element $v \neq u \in X$ such that $u \sqsubseteq v$ and there is no $w \in X$ with $w \neq u, w \neq v$, and $u \sqsubseteq w \sqsubseteq v$, i.e., v follows u in X with no other elements in between. A *maximal* element has no successor. A *meet irreducible* element is an element $u \in X$ such that for any $v, w \in X$, if $u = v \sqcap w$ then $u = v$ or $u = w$. A meet irreducible has at most one successor.

Given two partial orders $\langle X, \sqsubseteq \rangle$ and $\langle Y, \preceq \rangle$, an *embedding* of X into Y is a pair of functions $f : X \to Y$ and $g : (Y \times Y) \to \{0, 1\}$, which may have some of the following properties for all $u, v \in X$:

$$u \sqsubseteq v \Rightarrow f(u) \preceq f(v) \quad (1)$$

$$\mathit{defined}(u \sqcup v) \Rightarrow g(f(u), f(v)) = 1 \quad (2)$$

$$\neg \mathit{defined}(u \sqcup v) \Rightarrow g(f(u), f(v)) = 0 \quad (3)$$

$$u \sqcup v = w \Leftrightarrow f(u) \curlyvee f(v) = f(w) \quad (4)$$

With property (1), the embedding is said to preserve order; with property (2), it preserves success; with property (3), it preserves failure; and with property (4), it preserves joins.

2 Bit-vector encoding

Intuitively, taking the join of two types in a type hierarchy is like taking the intersection of two sets. Types often represent sets of possible values, and the type represented by the join really does represent the intersection of the sets that formed the input. So it seems natural to embed a partial order of types $\langle X, \sqsubseteq \rangle$ into a partial order (in fact, a lattice) of sets $\langle Y, \preceq \rangle$, where Y is the power set of some set Z, and $\preceq$ is the superset relation $\supseteq$. Then join $\curlyvee$ is simply set intersection $\cap$. The embedding function g, which indicates whether a join exists, can be naturally defined by $g(f(u), f(v)) = 0$ if and only if $f(u) \cap f(v) = \emptyset$. It remains to choose the underlying set Z and embedding function f.

Aït-Kaci et al. (1989) developed what has become the standard technique of this type. They set Z to be the set of all meet irreducible elements in X; and $f(u) = \{v \in Z | v \sqsupseteq u\}$, that is, the meet irreducible elements greater than or equal to u. The resulting embedding preserves order, success, failure, and joins. If Z is chosen to be the maximal elements of X instead, then join preservation is lost but the embedding still preserves order, success, and failure. The sets can be represented efficiently by vectors of bits. We hope to minimize the size of the largest set $f(\bot)$, which determines the vector length.

It follows from the work of Markowsky (1980) that the construction of Aït-Kaci et al. is optimal among encodings that use sets with intersection for meet and empty set for failure: with Y defined as the power set of some set Z, $\sqsubseteq$ as $\supseteq$, $\sqcup$ as $\cap$, and $g(f(u), f(v)) = 0$ if and only if $f(u) \cap f(v) = \emptyset$, then the smallest Z that will preserve order, success, failure, and joins is the set of all meet irreducible elements of X. No shorter bit vectors are possible.

We construct shorter bit vectors by modifying the definition of g, so that the minimality results

no longer apply. In the following discussion we present first an intuitive and then a technical description of our approach.

2.1 Intuition from Bloom filters

Vectors generated by the above construction tend to be quite sparse, or if not sparse, at least boring. Consider a meet semilattice containing only the bottom element $\perp$ and n maximal elements all incomparable to each other. Then each bit vector would consist of either all ones, or all zeroes except for a single one. We would thus be spending n bits to represent a choice among $n + 1$ alternatives, which should fit into a logarithmic number of bits. The meet semilattices that occur in practice are more complicated than this example, but they tend to contain things like it as a substructure. With the traditional bit vector construction, each of the maximal elements consumes its own bit, even though those bits are highly correlated.

The well-known technique called Bloom filtering (Bloom, 1970) addresses a similar issue. There, it is desired to store a large array of bits subject to two considerations. First, most of the bits are zeroes. Second, we are willing to accept a small proportion of one-sided errors, where every query that should correctly return one does so, but some queries that should correctly return zero might actually return one instead.

The solution proposed by Bloom and widely used in the decades since is to map the entries in the large bit array pseudorandomly (by means of a hash function) into the entries of a small bit array. To store a one bit we find its hashed location and store it there. If we query a bit for which the answer should be zero but it happens to have the same hashed location as another query with the answer one, then we return a one and that is one of our tolerated errors.

To reduce the error rate we can elaborate the construction further: with some fixed k, we use k hash functions to map each bit in the large array to several locations in the small one. Figure 1 illustrates the technique with $k = 3$. Each bit has three hashed locations. On a query, we check all three; they must all contain ones for the query to return a one. There will be many collisions of individual hashed locations, as shown; but the chances are good that when we query a bit we did not intend to store in the filter, at least one of its hashed locations will still be empty, and so the query will

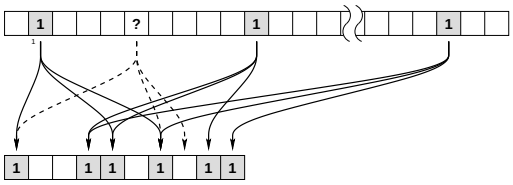

Figure 1: A Bloom filter

return zero. Bloom describes how to calculate the optimal value of k, and the necessary length of the hashed array, to achieve any desired bound on the error rate. In general, the hashed array can be much smaller than the original unhashed array (Bloom, 1970).

Classical Bloom filtering applied to the sparse vectors of the embedding would create some percentage of incorrect join results, which would then have to be handled by other techniques. Our work described here combines the idea of using k hash functions to reduce the error rate, with perfect hashes designed in a precomputation step to bring the error rate to zero.

2.2 Modified failure detection

In the traditional bit vector construction, types map to sets, join is computed by intersection of sets, and the empty set corresponds to failure (where no join exists). Following the lead of Bloom filters, we change the embedding function $g(f(u), f(v))$ to be 0 if and only if $|f(u) \cap f(v)| \leq \lambda$ for some constant λ. With $\lambda = 0$ this is the same as before. Choosing greater values of λ allows us to re-use set elements in different parts of the type hierarchy while still avoiding collisions.

Figure 2 shows an example meet semilattice. In the traditional construction, to preserve joins we must assign one bit to each of the meet-irreducible elements $\{d, e, f, g, h, i, j, k, l, m\}$, for a total of ten bits. But we can use eight bits and still preserve joins by setting $g(f(u), f(v)) = 0$ if and only if $|f(u) \cap f(v)| \leq \lambda = 1$, and f as follows.

$$
\begin{array}{cc}
f(\perp) = \{1,2,3,4,5,6,7,8\} & \\
f(a) = \{1,2,3,4,5\} & \\
f(b) = \{1,6,7,8\} & f(c) = \{1,2,3\} \\
f(d) = \{2,3,4,5\} & f(e) = \{1,6\} \\
f(f) = \{1,7\} & f(g) = \{1,8\} \\
f(h) = \{6,7\} & f(i) = \{6,8\} \\
f(j) = \{1,2\} & f(k) = \{1,3\} \\
f(l) = \{2,3\} & f(m) = \{2,3,4\}
\end{array}
\tag{5}
$$

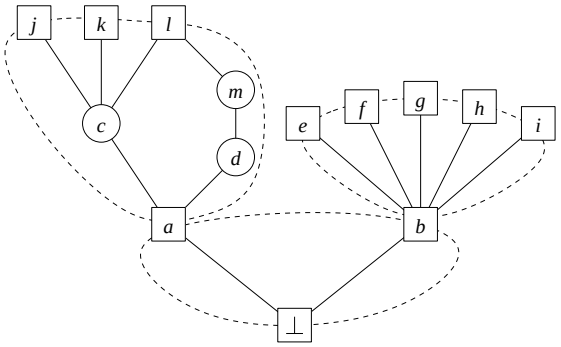

Figure 2: An example meet semilattice; $\perp$ is the most general type.

As a more general example, consider the very simple meet semilattice consisting of just a least element $\perp$ with n maximal elements incomparable to each other. For a given λ we can represent this in b bits by choosing the smallest b such that $\binom{b}{\lambda+1} \geq n$ and assigning each maximal element a distinct choice of the bits. With optimal choice of λ, b is logarithmic in n.

2.3 Modules

As Aït-Kaci et al. (1989) described, partial orders encountered in practice often resemble trees. Both their technique and ours are at a disadvantage when applied to large trees; in particular, if the bottom of the partial order has successors which are not joinable with each other, then those will be assigned large sets with little overlap, and bits in the vectors will tend to be wasted.

To avoid wasting bits, we examine the partial order X in a precomputation step to find the *modules*, which are the smallest upward-closed subsets of X such that for any $x \in X$, if x has at least two joinable successors, then x is in a module. This is similar to ALE's definition of module (Penn, 1999), but not the same. The definition of Aït-Kaci et al. (1989) also differs from ours. Under our definition, every module has a unique least element, and not every type is in a module. For instance, in Figure 2, the only module has a as its least element. In the ERG's type hierarchy, there are 11 modules, with sizes ranging from 10 to 1998 types.

To find the join of two types in the same module, we find the intersection of their encodings and check whether it is of size greater than λ. If the types belong to two distinct modules, there is no join. For the remaining cases, where at least one of the types lacks a module, we observe that the module bottoms and non-module types form a tree, and the join can be computed in that tree. If x is a type in the module whose bottom is y, and z has no module, then $x \sqcup z = y \sqcup z$ unless $y \sqcup z = y$ in which case $x \sqcup z = x$; so it only remains to compute joins within the tree. Our implementation does that by table lookup. More sophisticated approaches could be appropriate on larger trees.

3 Set programming

Ideally, we would like to have an efficient algorithm for finding the best possible encoding of any given meet semilattice. The encoding can be represented as a collection of sets of integers (representing bit indices that contain ones), and an optimal encoding is the collection of sets whose overall union is smallest subject to the constraint that the collection forms an encoding at all. This combinatorial optimization problem is a form of *set programming*; and set programming problems are widely studied. We begin by defining the form of set programming we will use.

Definition 1 *Choose set variables $S_1, S_2, \ldots, S_n$ to minimize $b = |\bigcup_{i=1}^{n} S_i|$ subject to some constraints of the forms $|S_i| \geq r_i$, $S_i \subseteq S_j$, $S_i \not\supseteq S_j$, $|S_i \cap S_j| \leq \lambda$, and $S_i \cap S_j = S_k$. The constant λ is the same for all constraints. Set elements may be arbitrary, but we generally assume they are the integers $\{1 \ldots b\}$ for convenience.*

The reduction of partial order representation to set programming is clear: we create a set variable for every type, force the maximal types' sets to contain at least $\lambda + 1$ elements, and then use subset to enforce that every type is a superset of all its successors (preserving order and success). We limit the maximum intersection of incomparable types to preserve failure. To preserve joins, if that property is desired, we add a constraint $S_i \not\supseteq S_j$ for every pair of types $x_i \not\sqsubseteq x_j$ and one of the form $S_i \cap S_j = S_k$ for every x_i, x_j, x_k such that $x_i \sqcup x_j = x_k$..

Given a constraint satisfaction problem like this one, we can ask two questions: is there a *feasible* solution, assigning values to the variables so all constraints are satisfied; and if so what is the *optimal* solution, producing the best value of the objective while remaining feasible? In our problem, there is always a feasible solution we can find by the *generalized Aït-Kaci et al. construction* (GAK), which consists of assigning λ bits

shared among all types; adding enough unshared new bits to maximal elements to satisfy cardinality constraints; adding one new bit to each non-maximal meet irreducible type; and propagating all the bits down the hierarchy to satisfy the subset constraints. Since the GAK solution is feasible, it provides a useful upper bound on the result of the set programming.

Ongoing research on set programming has produced a variety of software tools for solving these problems. However, at first blush our instances are much too large for readily-available set programming tools. Grammars like ERG contain thousands of types. We use binary constraints between every pair of types, for a total of millions of constraints—and these are variables and constraints over a domain of sets, not integers or reals. General-purpose set programming software cannot handle such instances.

3.1 Simplifying the instances

First of all, we only use minimum cardinality constraints $|S_i| \geq r_i$ for maximal types; and every $r_i \geq \lambda + 1$. Given a feasible bit assignment for a maximal type with more than r_i elements in its set S_i, we can always remove elements until it has exactly r_i elements, without violating the other constraints. As a result, instead of using constraints $|S_i| \geq r_i$ we can use constraints $|S_i| = r_i$. Doing so reduces the search space.

Subset is transitive; so if we have constraints $S_i \subseteq S_j$ and $S_j \subseteq S_k$, then $S_i \subseteq S_k$ is implied and we need not specify it as a constraint. Similarly, if we have $S_i \subseteq S_j$ and $S_i \nsubseteq S_k$, then we have $S_j \nsubseteq S_k$. Furthermore, if S_i and S_j have maximum intersection λ, then any subset of S_i also has maximum intersection λ with any subset of S_k, and we need not specify those constraints either.

Now, let a *choke-vertex* in the partial order $\langle X, \sqsubseteq \rangle$ be an element $u \in X$ such that for every $v, w \in X$ where v is a successor of w and $u \sqsubseteq v$, we have $u \sqsubseteq w$. That is, any chain of successors from elements not after u to elements after u, must pass through u. Figure 2 shows choke-vertices as squares. We call these choke-vertices by analogy with the graph theoretic concept of cut-vertices in the Hasse diagram of the partial order; but note that some vertices (like j and k) can be choke-vertices without being cut-vertices, and some vertices (like c) can be cut-vertices without being choke-vertices. Maximal and minimal elements are always choke-vertices.

Choke-vertices are important because the optimal bit assignment for elements after a choke-vertex u is almost independent of the bit assignment elsewhere in the partial order. Removing the redundant constraints means there are no constraints between elements after u and elements before, or incomparable with, u. All constraints across u must involve u directly. As a result, we can solve a smaller instance consisting of u and everything after it, to find the minimal number of bits r_u for representing u. Then we solve the rest of the problem with a constraint $|S_u| = r_u$, excluding all partial order elements after u, and then combine the two solutions with any arbitrary bijection between the set elements assigned to u in each solution. Assuming optimal solutions to both sub-problems, the result is an optimal solution to the original problem.

3.2 Splitting into components

If we cut the partial order at every choke-vertex, we reduce the huge and impractical encoding problem to a collection of smaller ones. The cutting expresses the original partial order as a tree of *components*, each of which corresponds to a set programming instance. Components are shown by the dashed lines in Figure 2. We can find an optimal encoding for the entire partial order by optimally encoding the components, starting with the leaves of that tree and working our way back to the root.

The division into components creates a collection of set programming instances with a wide range of sizes and difficulty; we examine each instance and choose appropriate techniques for each one. Table 1 summarizes the rules used to solve an instance, and shows the number of times each rule was applied in a typical run with the modules extracted from ERG, a ten-minute timeout, and each λ from 0 to 10.

In many simple cases, GAK is provably optimal. These include when $\lambda = 0$ regardless of the structure of the component; when the component consists of a bottom and zero, one, or two non-joinable successors; and when there is one element (a *top*) greater than all other elements in the component. We can easily recognize these cases and apply GAK to them.

Another important special case is when the

Condition	Succ.	Fail.	Method
$\lambda = 0$	216		GAK (optimal)
$\exists$ top	510		GAK (optimal)
2 successors	850		GAK (optimal)
3 or 4 successors	70		exponential variable
only ULs	420		b-choose-$(\lambda+1)$ special case
before UL removal	251	59	`ic_sets`
after UL removal	9	50	`ic_sets`
remaining	50		GAK

Table 1: Rules for solving an instance in the ERG

component consists of a bottom and some number k of pairwise non-joinable successors, and the successors all have required cardinality $\lambda + 1$. Then the optimal encoding comes from finding the smallest b such that $\binom{b}{\lambda+1}$ is at least k, and giving each successor a distinct combination of the b bits.

3.3 Removing unary leaves

For components that do not have one of the special forms described above, it becomes necessary to solve the set programming problem. Some of our instances are small enough to apply constraint solving software directly; but for larger instances, we have one more technique to bring them into the tractable range.

Definition 2 *A* unary leaf *(UL) is an element x in a partial order $\langle X, \sqsubseteq \rangle$ such that x is maximal and x is the successor of exactly one other element.*

ULs are special because their set programming constraints always take a particular form: if x is a UL and a successor of y, then the constraints on its set S_x are exactly that $|S_x| = \lambda + 1$, $S_x \subseteq S_y$, and S_x has intersection of size at most λ with the set for any other successor of y. Other constraints disappear by the simplifications described earlier.

Furthermore, ULs occur frequently in the partial orders we consider in practice; and by increasing the number of sets in an instance, they have a disproportionate effect on the difficulty of solving the set programming problem. We therefore implement a special solution process for instances containing ULs: we remove them all, solve the resulting instance, and then add them back one at a time while attempting to increase the overall number of elements as little as possible.

This process of removing ULs, solving, and adding them back in, may in general produce suboptimal solutions, so we use it only when the solver cannot find a solution on the full-sized problem. In practical experiments, the solver generally either produces an optimal or very nearly optimal solution within a time limit on the order of ten minutes; or fails to produce a feasible solution at all, even with a much longer limit. Testing whether it finds a solution is then a useful way to determine whether UL removal is worthwhile.

Recall that in an instance consisting of k ULs and a bottom, an optimal solution consists of finding the smallest b such that $\binom{b}{\lambda+1}$ is at least k; that is the number of bits for the bottom, and we can choose any k distinct subsets of size $\lambda + 1$ for the ULs. Augmenting an existing solution to include additional ULs involves a similar calculation.

To add a UL x as the successor of an element y without increasing the total number of bits, we must find a choice of $\lambda + 1$ of the bits already assigned to y, sharing at most λ bits with any of y's other successors. Those successors are in general sets of arbitrary size, but all that matters for assigning x is how many subsets of size $\lambda + 1$ they already cover. The UL can use any such subset not covered by an existing successor of y. Our algorithm counts the subsets already covered, and compares that with the number of choices of $\lambda+1$ bits from the bits assigned to y. If enough choices remain, we use them; otherwise, we add bits until there are enough choices.

3.4 Solving

For instances with a small number of sets and relatively large number of elements in the sets, we use an *exponential variable* solver. This encodes the set programming problem into integer programming. For each element $x \in \{1, 2, \ldots, b\}$, let $c(x) = \{i | x \in S_i\}$; that is, $c(x)$ represents the indices of all the sets in the problem that contain the element x. There are $2^n - 1$ possible values of $c(x)$, because each element must be in at least one set. We create an integer variable for each of those values. Each element is counted once, so the sum of the integer variables is b. The constraints translate into simple inequalities on sums of the variables; and the system of constraints can be solved with standard integer programming techniques. After solving the integer programming problem we can then assign elements arbitrarily

to the appropriate combinations of sets.

Where applicable, the exponential variable approach works well, because it breaks all the symmetries between set elements. It also continues to function well even when the sets are large, since nothing in the problem directly grows when we increase b. The wide domains of the variables may be advantageous for some integer programming solvers as well. However, it creates an integer programming problem of size exponential in the number of sets. As a result, it is only applicable to instances with a very few set variables.

For more general set programming instances, we feed the instance directly into a solver designed for such problems. We used the ECL^{i}PS^e logic programming system (Cisco Systems, 2008), which offers several set programming solvers as libraries, and settled on the ic_sets library. This is a straightforward set programming solver based on containment bounds. We extended the solver by adding a lightweight not-subset constraint, and customized heuristics for variable and value selection designed to guide the solver to a feasible solution as soon as possible. We choose variables near the top of the instance first, and prefer to assign values that share exactly λ bits with existing assigned values. We also do limited symmetry breaking, in that whenever we assign a bit not shared with any current assignment, the choice of bit is arbitrary so we assume it must be the lowest-index bit. That symmetry breaking speeds up the search significantly.

The present work is primarily on the benefits of nonzero λ, and so a detailed study of general set programming techniques would be inappropriate; but we made informal tests of several other set-programming solvers. We had hoped that a solver using containment-lexicographic hybrid bounds as described by Sadler and Gervet (Sadler and Gervet, 2008) would offer good performance, and chose the ECL^{i}PS^e framework partly to gain access to its ic_hybrid_sets implementation of such bounds. In practice, however, ic_hybrid_sets gave consistently worse performance than ic_sets (typically by an approximate factor of two). It appears that in intuitive terms, the lexicographic bounds rarely narrowed the domains of variables much until the variables were almost entirely labelled anyway, at which point containment bounds were almost as good; and meanwhile the increased overhead of maintaining the extra bounds slowed down the entire process to more than compensate for the improved propagation. We also evaluated the Cardinal solver included in ECL^{i}PS^e, which offers stronger propagation of cardinality information; it lacked other needed features and seemed no more efficient than ic_sets. Among these three solvers, the improvements associated with our custom variable and value heuristics greatly outweighed the baseline differences between the solvers; and the differences were in optimization time rather than quality of the returned solutions.

Solvers with available source code were preferred for ease of customization, and free solvers were preferred for economy, but a license for ILOG CPLEX (IBM, 2008) was available and we tried using it with the natural encoding of sets as vectors of binary variables. It solved small instances to optimality in time comparable to that of ECL^{i}PS^e. However, for medium to large instances, CPLEX proved impractical. An instance with n sets of up to b bits, dense with pairwise constraints like subset and maximum intersection, requires $\Theta(n^2b)$ variables when encoded into integer programming in the natural way. CPLEX stores a copy of the relaxed problem, with significant bookkeeping information per variable, for every node in the search tree. It is capable of storing most of the tree in compressed form on disk, but in our larger instances even a single node is too large; CPLEX exhausts memory while loading its input. The ECL^{i}PS^e solver also stores each set variable in a data structure that increases linearly with the number of elements, so that the size of the problem as stored by ECL^{i}PS^e is also $\Theta(n^2b)$; but the constant for ECL^{i}PS^e appears to be much smaller, and its search algorithm stores only incremental updates (with nodes per set instead of per element) on a stack as it explores the tree. As a result, the ECL^{i}PS^e solver can process much larger instances than CPLEX without exhausting memory.

Encoding into SAT would allow use of the sophisticated solvers available for that problem. Unfortunately, cardinality constraints are notoriously difficult to encode in Boolean logic. The obvious encoding of our problem into CNFSAT would require $O(n^2b\lambda)$ clauses and variables. Encodings into Boolean variables with richer constraints than CNFSAT (we tried, for instance, the SICStus Prolog clp(FD) implementation (Carlsson et al., 1997)) generally exhausted memory on much smaller instances than those handled by the set-

Module	n	b_0	λ	b_λ
`mrs_min`	10	7	0	7
`conj`	13	8	1	7
`list`	27	15	1	11
`local_min`	27	21	1	10
`cat_min`	30	17	1	14
`individual`	33	15	0	15
`head_min`	247	55	0	55
`*sort*`	247	129	3	107
`synsem_min`	612	255	0	255
`sign_min`	1025	489	3	357
`mod_relation`	1998	1749	6	284
entire ERG	4305	2788	140	985

Table 2: Best encodings of the ERG and its modules: n is number of types, b_0 is vector length with $\lambda = 0$, and λ is parameter that gives the shortest vector length b_λ.

variable solvers, while offering no improvement in speed.

4 Evaluation

Table 2 shows the size of our smallest encodings to date for the entire ERG without modularization, and for each of its modules. These were found by running the optimization process of the previous section on Intel Xeon servers with a timeout of 30 minutes for each invocation of the solver (which may occur several times per module). Under those conditions, some modules take a long time to optimize—as much as two hours per tested value of λ for `sign_min`. The Xeon's hyperthreading feature makes reproducibility of timing results difficult, but we found that results almost never improved with additional time allowance beyond the first few seconds in any case, so the practical effect of the timing variations should be minimal.

These results show some significant improvements in vector length for the larger modules. However, they do not reveal the entire story. In particular, the apparent superiority of $\lambda = 0$ for the `synsem_min` module should not be taken as indicating that no higher λ could be better: rather, that module includes a very difficult set programming instance on which the solver failed and fell back to GAK. For the even larger modules, nonzero λ proved helpful despite solver failures, because of the bits saved by UL removal. UL removal is clearly a significant advantage, but only for the modules where the solver is failing anyway. One important lesson seems to be that further work on set programming solvers would be beneficial: any future more capable set programming solver could be applied to the unsolved instances and would be expected to save more bits.

Encoding	length	time	space
Lookup table	n/a	140	72496
Modular, best λ	0–357	321	203
Modular, $\lambda = 0$	0–1749	747	579
Non-mod, $\lambda = 0$	2788	4651	1530
Non-mod, $\lambda = 1$	1243	2224	706
Non-mod, $\lambda = 2$	1140	2008	656
Non-mod, $\lambda = 9$	1069	1981	622
Non-mod, $\lambda = 140$	985	3018	572

Table 3: Query performance. Vector length in bits, time in milliseconds, space in Kbytes.

Table 3 and Figure 3 show the performance of the join query with various encodings. These results are from a simple implementation in C that tests all ordered pairs of types for joinability. As well as testing the non-modular ERG encoding for different values of λ, we tested the modularized encoding with $\lambda = 0$ for all modules (to show the effect of modularization alone) and with λ chosen per-module to give the shortest vectors. For comparison, we also tested a simple lookup table. The same implementation sufficed for all these tests, by means of putting all types in one module for the non-modular bit vectors or no types in any module for the pure lookup table. The times shown are milliseconds of user CPU time to test all join tests (roughly 18.5 million of them), on a non-hyperthreading Intel Pentium 4 with a clock speed of 2.66GHz and 1G of RAM, running Linux. Space consumption shown is the total amount of dynamically-allocated memory used to store the vectors and lookup table.

The non-modular encoding with $\lambda = 0$ is the basic encoding of Aït-Kaci et al. (1989). As Table 3 shows, we achieved more than a factor of two improvement from that, in both time and vector length, just by setting $\lambda = 1$. Larger values offered further small improvements in length up to $\lambda = 140$, which gave the minimum vector length of 985. That is a shallow minimum; both $\lambda = 120$ and $\lambda = 160$ gave vector lengths of 986, and the length slowly increased with greater λ.

However, the fastest bit-count on this architec-

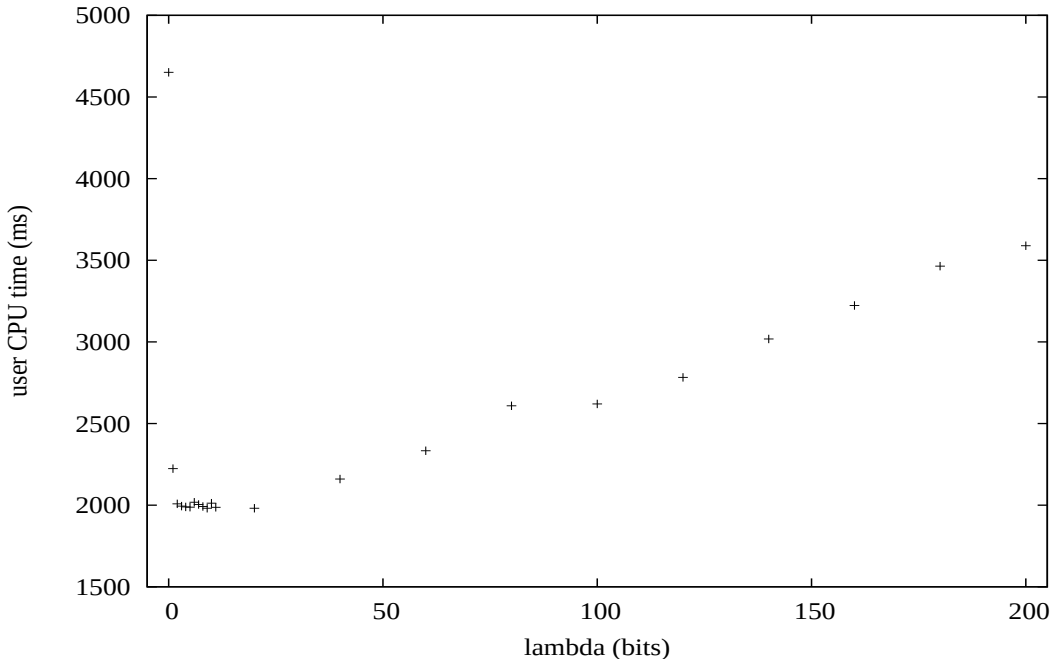

Figure 3: Query performance for the ERG without modularization.

ture, using a technique first published by Wegner (1960), requires time increasing with the number of nonzero bits it counts; and a similar effect would appear on a word-by-word basis even if we used a constant-time per-word count. As a result, there is a time cost associated with using larger λ, so that the fastest value is not necessarily the one that gives the shortest vectors. In our experiments, $\lambda = 9$ gave the fastest joins for the non-modular encoding of the ERG. As shown in Figure 3, all small nonzero λ gave very similar times.

Modularization helps a lot, both with $\lambda = 0$, and when we choose the optimal λ per module. Here, too, the use of optimal λ improves both time and space by more than a factor of two. Our best bit-vector encoding, the modularized one with per-module optimal λ, is only a little less than half the speed of the lookup table; and this test favours the lookup table by giving it a full word for every entry (no time spent shifting and masking bits) and testing the pairs in a simple two-level loop (almost purely sequential access).

5 Conclusion

We have described a generalization of conventional bit vector concept lattice encoding techniques to the case where all vectors with λ or fewer one bits represent failure; traditional encodings are the case $\lambda = 0$. Increasing λ can reduce the overall storage space and improve speed.

A good encoding requires a kind of perfect hash, the design of which maps naturally to constraint programming over sets of integers. We have described a practical framework for solving the instances of constraint programming thus created, in which we can apply existing or future constraint solvers to the subproblems for which they are best suited; and a technique for modularizing practical type hierarchies to get better value from the bit vector encodings. We have evaluated the resulting encodings on the ERG's type system, and examined the performance of the associated unification test. Modularization, and the use of nonzero λ, each independently provide significant savings in both time and vector length.

The modified failure detection concept suggests several directions for future work, including evaluation of the new encodings in the context of a large-scale HPSG parser; incorporation of further developments in constraint solvers; and the possibility of approximate encodings that would permit one-sided errors as in traditional Bloom filtering.

References

Hassan Aït-Kaci, Robert S. Boyer, Patrick Lincoln, and Roger Nasr. 1989. Efficient implementation of lattice operations. *ACM Transactions on Programming Languages and Systems*, 11(1):115–146, January.

Burton H. Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July.

Ulrich Callmeier. 2000. PET – a platform for experimentation with efficient HPSG processing techniques. *Natural Language Engineering*, 6(1):99–107.

Mats Carlsson, Greger Ottosson, and Björn Carlson. 1997. An open-ended finite domain constraint solver. In H. Glaser, P. Hartel, and H. Kucken, editors, *Programming Languages: Implementations, Logics, and Programming*, volume 1292 of *Lecture Notes in Computer Science*, pages 191–206. Springer-Verlag, September.

Cisco Systems. 2008. ECL^{i}PS^e 6.0. Computer software. Online `http://eclipse-clp.org/`.

Ann Copestake and Dan Flickinger. 2000. An open-source grammar development environment and broad-coverage English grammar using HPSG. In *Proceedings of the Second Conference on Language Resources and Evaluation (LREC 2000)*.

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

IBM. 2008. ILOG CPLEX 11. Computer software.

George Markowsky. 1980. The representation of posets and lattices by sets. *Algebra Universalis*, 11(1):173–192.

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

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

Gerald Penn. 1999. An optimized prolog encoding of typed feature structures. In D. De Schreye, editor, *Logic programming: proceedings of the 1999 International Conference on Logic Programming (ICLP)*, pages 124–138.

Gerald Penn. 2002. Generalized encoding of description spaces and its application to typed feature structures. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics (ACL 2002)*, pages 64–71.

Andrew Sadler and Carmen Gervet. 2008. Enhancing set constraint solvers with lexicographic bounds. *Journal of Heuristics*, 14(1).

David Talbot and Miles Osborne. 2007. Smoothed Bloom filter language models: Tera-scale LMs on the cheap. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, pages 468–476.

Peter Wegner. 1960. A technique for counting ones in a binary computer. *Communications of the ACM*, 3(5):322.