Appeared in TOPLAS 11-1-1989

Efficient Implementation of Lattice Operations[†]

Hassan Aït-Kaci
Robert Boyer[‡]
Patrick Lincoln
Roger Nasr

July 1988

MCC, ACA Program
3500 West Balcones Center Drive
Austin, TX 78759

Abstract

Lattice operations such as greatest lower bound (GLB), least upper bound (LUB), and relative complementation (BUTNOT) are becoming more and more important in programming languages supporting object inheritance. We present a general technique for the efficient implementation of such operations based on an encoding method. The effect of the encoding is to plunge the given ordering into a boolean lattice of binary words, leading to an almost constant time complexity of the lattice operations. A first method is described based on a transitive closure approach. Then, a more space-efficient method minimizing code-word length is described. Finally, a powerful grouping technique called *modulation* is presented which drastically reduces code space while keeping all three lattice operations highly efficient. This technique takes into account idiosyncrasies of the topology of the poset being encoded which are quite likely to occur in practice. All methods are formally justified. We see this work as an original contribution towards using semantic (*vz.*, in this case, taxonomic) information in the engineering pragmatics of storage and retrieval of (*vz.*, partially or quasi-ordered) information.

Keywords: Lattice operations, Inheritance, Partially-ordered objects, Symbolic computation, Information storage and retrieval.

[†]This article is a substantially revised and extended version of [7].

[‡]R. Boyer's present address is: Department of Computer Sciences, University of Texas, Austin, Texas 78712.

> There is a greatest cardinal in each type, namely, the cardinal number of the whole of the type; but this is always surpassed by the cardinal number of the next type, since, if α is the cardinal number of one type, that of the next type is 2^α, which, as Cantor has shown, is always greater than α.
>
> BERTRAND RUSSELL, *The Theory of Types.*

1 Introduction

Object inheritance is semantically and pragmatically useful. Novel programming languages are being designed which allow the user to specify type hierarchies. Examples of these are class inheritance in Common Loops [9] and SmallTalk [12], flavor inheritance in ZetaLisp [20], subsorts in Theorem Proving [18, 19], type checking in Amber [10], Galileo [2], and more recently object-oriented programming formulated as order-sorted algebraic abstract data-typing [11, 17], and David McAllester's work on boolean class expressions [14].

In general, objects which are instance of classes organized in a partial order, are manipulated as constraint expressions specifying conjunction, disjunction, or exclusion of certain class properties. Languages which support multiple inheritance (*i.e.*, where a class may have more than one superclass) such as ZetaLisp [20] or Smalltalk [12] use an *ad hoc* solution for combining class properties which depend on the temporal total order in which the classes are defined or appear in an expression. This is clearly semantically unclean and pragmatically hazardous. Other proposals [3, 19, 14] have formalized the concept of class inheritance in lattice-theoretic terms. This captures the essential properties useful in practice for object-oriented languages, and allows a better handling of class expressions for efficient implementation. *E.g.*, given a taxonomy of objects—a *subsumption* partial ordering—how expensive is it to compute the greatest common lower bounds of two objects?

In this paper, we present a method which can be used as a static (*i.e.*, at compilation time) procedure based on an idea of carrying the order-theoretic information of the object taxonomy into a homomorphic image where GLB computation is efficient. Although the method is general as far as which codes are used, this study focuses on the code space of (unbounded) binary words, which is the canonical boolean lattice under the cartesian extension of the $0 < 1$ ordering. The computation of greatest lower bounds, for example, is thus reduced to a binary *and*.

We developed this technique implementing the particular mechanism of object inheritance of LogIn [6] and further extended in Life [5]. Another promising prospect of relevance for the techniques we describe seems to be the fast implementation of other constraint programming models strongly related to ours such as [13, 15, 16].

Section 2 states the problem in general terms. We first focus on the GLB operation, the most commonly used operation. Given that in arbitrary posets such an operation is not necessarily defined, Section 3 recalls a simple semi-lattice embedding construction to palliate this. Then, a first method based on transitive closure is presented in Section 4. This method is altered to yield a more space-efficient encoding algorithm in Section 5. In Section 6, other lattice operations are considered; *i.e.*, how the method may be extended to support object disjunction and complementation. Finally, Section 7 elaborates a more sophisticated technique called *modulation*. The idea is that in pratice related elements come in "blobs" (which we call modules) such that it is possible to encode elements locally within a module, these being themselves partially-ordered and encoded. A comparative study of all three methods on large posets illustrates the gains in

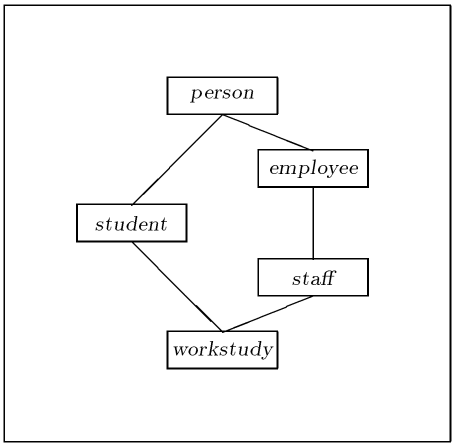


Figure 1: A poset with multiple inheritance

function $GLB(s, t : element)$ **returns** $: element$;
  begin
  $LBs \leftarrow \{x \in \Sigma \mid x \leq s\}$;
  $LBt \leftarrow \{x \in \Sigma \mid x \leq t\}$;
  $CLB \leftarrow LBs \cap LBt$;
  return $max(CLB)$
  end

Figure 2: The "brute-force" GLB algorithm

space and time incurred.

2 The Problem

Consider the object taxonomy—henceforth referred to as a (*object*) *poset*—of Figure 1. Let us suppose that some object is determined to be both of type *student* and type *employee*; *i.e.*, it inherits from both types. Object inheritance as type coercion is thus the operation of finding the greatest type, with respect to the subsumption ordering, which is a subtype of both *student* and *employee*; in this case, *workstudy*.

Given that the GLB of any two elements exists and is unique—*i.e.*, that the poset is a lower semi-lattice (*LSL*)—we need an algorithm to compute it. A general, albeit naïve, algorithm to compute the GLB of two elements of a finite LSL Σ, is given in Figure 2. That is, the GLB of two elements s and t is *the* greatest element of the set of common lower bounds of s and t in Σ. This algorithm is probably not the most efficient one could find. However, it is undoubtedly correct.

Let us suppose now that there exists a LSL $\mathcal{L}, \sqsubseteq, \sqcap$ for which we know how to compute GLB's efficiently. Now, given some LSL $\Sigma, \leq, \wedge$, let us also suppose that there exists a function γ from

Σ to $\mathcal{L}$ such that, for any two elements s and t in Σ:

$$\gamma(s \wedge t) \;=\; \gamma(s) \sqcap \gamma(t) \tag{1}$$

i.e., γ is a LSL homomorphism. Finally, let us also assume that the function γ is invertible; *i.e.*, that there exists a function γ^{-1} from $\mathcal{L}$ to Σ such that, for any s in Σ:

$$\gamma^{-1}(\gamma(s)) \;=\; s \tag{2}$$

Then, a way of computing the GLB of two elements s and t in the semi-lattice Σ is to combine Equations (1) and (2):

$$s \wedge t \;=\; \gamma^{-1}(\gamma(s) \sqcap \gamma(t)) \tag{3}$$

More precisely, Equation (3) is an efficient way to compute GLB's in Σ only if the function γ and its inverse are also efficiently computable. And this is not an assumption we may realistically make. However, we need not really use Equation (3) literally. The function γ may be relatively expensive, as long as we can compute it statically. Indeed, a compile-time computation could thus compute all the γ-images of the elements in Σ, so that all the run-time GLB computations be carried out in $\mathcal{L}$, and only pay the price of computing the inverse image by γ of the ultimate result. This idea is invaluable for a language like LogIn [6] where run-time computation consists essentially of very high number of such GLB operations, and decoding with γ^{-1} is needed only to print out the result—obviously a small price to pay.

This is indeed the gist of the technique we are to propose. We suggest viewing such a γ function as a *code*, and compilation of the object inheritance operation thus becomes an encoding process.

Naturally, we need to explicate other wild assumptions made above, such as the LSL structure of Σ, and the decoding function γ^{-1}. We next dismiss the former by recalling a semi-lattice embedding. As for the latter, a simple decoding follows from the LSL embedding interpretation.

3 A Semi-Lattice Construction

The foregoing simple analysis relies on the assumption that the poset Σ must be a lower semi-lattice; *i.e.*, that a *unique* GLB exists in Σ for any two object symbols in Σ (given by the $\wedge$ operation). However, in practice this is not quite a reasonable assumption to make. Indeed, in order to maintain this assumption valid, as the size of the poset grows, there must be an exponential number of pairwise GLB's to be specified—clearly, an inappropriate demand on a programmer.

Instead, it would be simpler to embed a partially-ordered object set Σ which is not a LSL into the least such structure which contains it—up to some isomorphism. This embedding must preserve the order structure of Σ, and in particular, existing GLB's. Such an embedding must also be semantically sound, in that the operational logic it implements must be consistent—in our case, all or some consistent restrictions of propositional logic implemented as boolean codes and operations.

The idea is rather simple, and makes intuitive sense. Let us consider for example the poset of Figure 3. Objects $w_1, \ldots, w_k$ are both students and employees. However, there is not a common object symbol to designate the set of students and employees. Thus, taking the GLB of *student* and *employee* in this poset cannot be defined as a unique element of the poset. Nevertheless, it makes sense to say that the GLB of *student* and *employee* ought to be the *set*

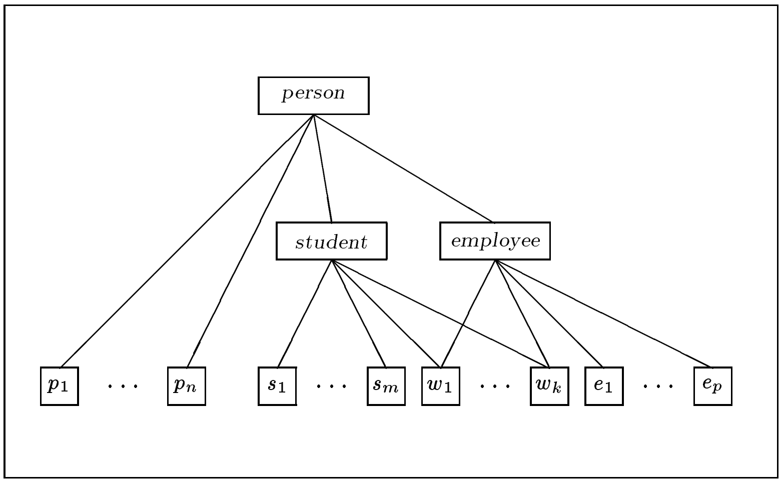


Figure 3: A poset which is not a semi-lattice

$\{w_1, \ \ldots \ , w_k\}$. This is precisely the effect that the following construction achieves. To our knowledge, this construction is not conventional. It is related to what is known as *completion by ideals* [8], and detailed in the particular following form in [3] and [4].

In what follows, we make the assumption that the poset is finite.[1] First, we need some definitions.

The *restricted powerset* of a poset $S, \leq$ is the set $\mathbf{2}^{(S)}$of non-empty finite subsets of pairwise *incomparable* elements of S. Such subsets are called *cochains*—or, more figuratively, *crowns*—and are partially-ordered by a relation $\sqsubseteq$ defined by:

$$X \sqsubseteq Y \text{ \textit{iff} } \forall x \in X, \exists y \in Y, \ x \leq y.$$

Given a poset $S, \leq$, the *canonical injection* (written ι) from S into $\mathbf{2}^{(S)}$is the function which takes every element x of S into the singleton $\iota(x) = \{x\}$. This simple function has the nice property that :

$$\forall x \in S, \forall y \in S, \ \iota(x) \sqsubseteq \iota(y) \text{ \textit{iff} } x \leq y.$$

That is, ι is an order homomorphism. Given any subset X of S, we define its *maximal restriction* $\lceil X \rceil$ as the set of maximal elements of X. Clearly, $\lceil X \rceil$ is in $\mathbf{2}^{(S)}$, and defined for all subsets of a finite poset S. Given some element x of S, we note by $\underline{x}$ the subset of S of all lower bounds of x. That is,

$$\underline{x} = \{y \in S \mid y \leq x\}.$$

Then, for any two elements a, b in S, $\underline{a} \cap \underline{b}$ is the set of common lower bounds of a and b in S. Finally, the following binary operation $\sqcap$ can be defined on $\mathbf{2}^{(S)}$for any pair of subsets X, Y:

$$X \sqcap Y = \left\lceil \bigcup_{a \in X}^{b \in Y} \underline{a} \cap \underline{b} \right\rceil \tag{4}$$

[1]In fact, as shown in [3], such a construction can be performed for an infinite poset which is Noetherian—*i.e.*, one which does not contain infinite ascending chains.

and this operation is a GLB operation in $\mathbf{2}^{(S)}$.

As a result, $\mathbf{2}^{(S)}, \sqsubseteq, \sqcap$ is a lower semi-lattice. Furthermore, we observe that if two elements x and y in S already have a unique GLB z in S, it follows that:

$$\{x\} \sqcap \{y\} \;=\; \{z\}.$$

Hence, this construction is a structure embedding, in that it preserves the ordering and the GLB's when they exist in S. Now, we are justified to take the freedom of writing simply x rather than $\{x\}$ for any single element of an object poset Σ, and extend the poset to $\mathbf{2}^{(\Sigma)}$ the GLB preserving lower semi-lattice extension of Σ. And this is the "least" such possible structure, since if Σ is already a lower semi-lattice then it is isomorphic to its canonical injection into $\mathbf{2}^{(\Sigma)}$.

This construction is our formal justification of the fact that we need only deal with posets which are LSL's. In addition to being a universal embedding, this restricted powerset construction also permits the manipulation of disjunctive objects. This brings the problem of the decoding function γ^{-1}.

A convenient consequence of plunging the poset Σ in its restricted powerset $\mathbf{2}^{(\Sigma)}$ with ι, is that an inverse γ^{-1} for the the encoding function γ may be extracted as follows. Namely, γ^{-1} may be seen as the restriction of a function $\gamma_{\boldsymbol{s}}^{-1}$ from $\mathcal{L}$ to $\mathbf{2}^{(\Sigma)}$ such that, for any c in $\mathcal{L}$:

$$\gamma_{\boldsymbol{s}}^{-1}(c) \;=\; \lceil \{x \in \Sigma \mid \gamma(x) \sqsubseteq c\} \rceil \tag{5}$$

In words, $\gamma_{\boldsymbol{s}}^{-1}(c)$ is the set of maximal elements of Σ whose codes are less than c. What would then $\gamma_{\boldsymbol{s}}$, the inverse of $\gamma_{\boldsymbol{s}}^{-1}$ be? As we shall see, if $\mathcal{L}$ has the additional property of being a distributive lattice, γ may be extended to such a $\gamma_{\boldsymbol{s}}$ from $\mathbf{2}^{(\Sigma)}$ to $\mathcal{L}$, thus becoming always invertible.

4 Transitive Closure

First of all, let us remark that even if Σ, the object poset, is not a LSL, then the brute-force algorithm of Figure 2 remains correct. Then indeed, it yields the set of maximal common lower bounds of the input elements—the maximal simultaneously subsumed objects.

The two approaches described in the previous section are each obviously correct. However, as they stand, they are also impractical. The brute-force method would clearly lead to exponential computations, and it is not obvious how one could find an appropriate code function if one were to use the coding approach. Nevertheless, both ideas may be combined based on two observations:

1. Much redundant work in computing the sets of all lower bounds of poset elements could be performed statically, and saved for run-time use.

2. A simple code for a poset element could be a representation of the set of all its lower bounds.

The first of these remarks is achieved by computing a reflexive and transitive closure of the "immediately greater than" relation. The second, by using a well-known representation of sets as bit-vectors. The trick is that bit-vectors implement both $\mathbf{2}^{(\Sigma)}$ and $\mathcal{L}$, thus realizing the isomorphism between the two sets. This yields the simple encoding method explained next.

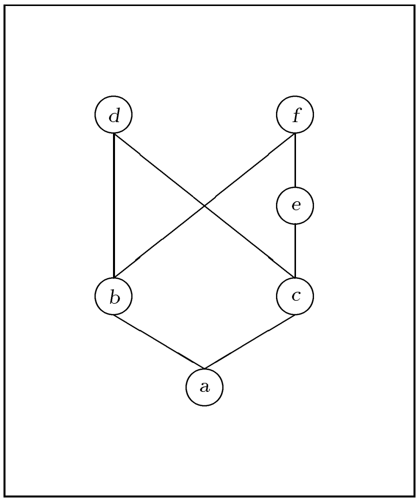


Figure 4: A Hasse diagram of a poset

	a	b	c	d	e	f
a	0	0	0	0	0	0
b	1	0	0	0	0	0
c	1	0	0	0	0	0
d	0	1	1	0	0	0
e	0	0	1	0	0	0
f	0	1	0	0	1	0

Figure 5: Bit-array of Hasse diagram

4.1 Example

Let us consider the 6-element poset of Figure 4. The "immediately greater than" relation covered by this ordering can be represented in a 6×6 array as shown in Figure 5. Each row contains 1's only in those columns headed by elements which are immediately less than the element heading the row; and it contains 0's otherwise. Thus, a row headed by an element x can be viewed as a characteristic boolean vector representation of the set of all the immediate strict lower bounds of x.

Now, taking the reflexive and transitive closure of the array in Figure 5, we obtain the array in Figure 6. It is obvious that each element's row in this array is a characteristic representation of the set of all its lower bounds. Referring back to the brute-force GLB algorithm of Figure 2, the intersection operation computing common lower bounds is hence reduced to a *binary and* operation on bit-vectors. We have thus come up with a code function γ which associates a 6-element bit-vector to any element x in the poset: the row of the boolean array representation of the reflexive and transitive closure of the "immediately greater than" relation.

To compute the GLB of, say, d and e, we take the *and* of 111100 and 101010, obtaining 101000 which is precisely the code of c, the GLB of d and e.

	a	b	c	d	e	f
a	1	0	0	0	0	0
b	1	1	0	0	0	0
c	1	0	1	0	0	0
d	1	1	1	1	0	0
e	1	0	1	0	1	0
f	1	1	1	0	1	1

Figure 6: Reflexive transitive closure

What happens if we try to compute the GLB of d and f? Taking the *and* of 111100 and 111011 yields 111000 which is the code of none of the poset elements. But using the decoding function γ_s^{-1} described by Equation (5), we obtain the set $\{b, c\}$ as wished; *i.e.*, the set of maximal common lower bounds of d and f.

Looking at Figure 6, we note that the column headed by a contains only 1's. Indeed, a is a *bottom* element in this poset, and thus every element is greater than it. Thus, a slightly more "compact" binary code word is obtained by dropping the bit of the bottom element, without any loss. We shall do that systematically, always coding the least element with 0.

4.2 Method

Given an $n \times n$ boolean array M, its reflexive and transitive closure M^* is given by:

$$M^* = \bigcup_{i=0}^{n} M^i \tag{6}$$

where the power operation is computed as matrix multiplication in the boolean ring of $n \times n$ bit-matrices. This yields a straightforward fixed-point algorithm which converges in at most $\log_2 n$ iterations.[2]

Hence, operationally, transitive closure is a well-known computation. Nevertheless, we shall present here a particular way of doing it. Our reasons will become clear later as we shall propose a slight modification of this idiosyncratic algorithm that would not be so immediate with other ways of implementing transitive closure. We now describe the algorithm, and in Section 4.3 we shall formally justify it as a transitive closure computation.

Since we are concerned with finding a compilation-time procedure, time efficiency of the coding method is not the prime worry. Instead, we may not mind paying the price of a slightly less efficient procedure if it leads to better run-time behavior.

Another way of computing such a transitive closure, is to work directly on the graph structure of the base relation. Starting at the bottom element, we can work upwards, layer by layer, assigning binary code words to each elements.[3] A layer is a set of incomparable elements—a cochain—computed from the previous layer as the set of all the immediate parents that cannot

[2]This is achieved by computing the sequence $N_0 = I \cup M$, $N_k = N_{k-1}^2, (k > 0)$, until $N_k = N_{k+1} = M^*$. Each iteration involves one matrix multiplication. Hence, this gives a possible overall time complexity $\mathcal{O}(n^\alpha \log n)$ where α was last known (by us) to be in the vicinity of 2.7, and according to rumors, keeps nibbling its way down. This method is based on techniques developed by Warshall-Strassen [1].

[3]Without loss of generality, we shall assume that such a least element always exists in Σ. If no such unique bottom exists, we can add one.

```
procedure AssignCode;
   begin
   p ← 0;
   γ(⊥) ← 0;
   L ← {⊥};
   while L ≠ ∅ do
      begin
      EncodeLayer(L);
      L ← NextLayer(L)
      end
   end
```

Figure 7: The AssignCode procedure

```
procedure EncodeLayer(L : cochain);
   begin
   for each x ∈ L do
      begin
      γ(x) ← 2^p ∨ ⋁_{y∈Children(x)} γ(y);
      p ← p + 1
      end
   end
```

Figure 8: The EncodeLayer procedure

be reached later. In this manner, the entire poset can be swept through. A simple way to encode a "node" in the graph, is to compute its code as the binary *or* of the code of its children *or*ed with 2^p, where p is the number of nodes visited since $\perp$.

Let's illustrate this procedure on the poset of Figure 4. We begin by assigning the code 0 to a. The first layer being $\{a\}$, we obtain the second layer as $\{b, c\}$. The code of b is computed as $\gamma(a) \vee 2^0 = 0 \vee 1 = 1$. Then, the code of c is obtained as $\gamma(a) \vee 2^1 = 0 \vee 2 = 10$. The layer obtained from $\{b, c\}$ is $\{d, e, f\} - \{f\} = \{d, e\}$. The reason why f is to be taken out is that it can be reached later. A simple test for such elements as f is that they do not have all their immediate children already coded. The code assigned to d is $\gamma(b) \vee \gamma(c) \vee 2^2 = 1 \vee 10 \vee 100 = 111$; and the code assigned to e is $\gamma(c) \vee 2^3 = 10 \vee 1000 = 1010$. Finally, the last layer is $\{f\}$, and the code of f is $\gamma(b) \vee \gamma(e) \vee 2^4 = 1 \vee 1010 \vee 10000 = 11011$.

One will immediately note that these codes are exactly the reversed versions of those given in the reflexive transitive array of Figure 6 where the least element's column has been dropped.

A pidgin-code algorithm for this encoding procedure is given as Figure 7, Figure 8, and Figure 9. It uses a global variable p counting the number of codes assigned. It assumes a function *Children* (*resp.*, *Parents*) that returns the set of elements immediately less (*resp.*, greater) than a given one, and a predicate *Coded* that is *true* of any already encoded element.

The next section elaborates on the correctness of the above algorithm as a transitive closure algorithm.

function $NextLayer(L : cochain)$ **returns** : $cochain$;
  begin
  $M \leftarrow \bigcup\{Parents(x) \mid x \in L\}$;
  return $M - \{x \in M \mid \exists y \in Children(x)$ **and** $\neg Coded(y)\}$
  end

Figure 9: The NextLayer instruction

4.3 Correctness

To simplify the proof, we shall reinstate the least element "column" in the computation of codes. That is, we shall prove the correctness of our *AssignCode* encoding procedure where $\gamma(\bot)$ and p are both initialized to 1 instead of 0.

First, it is clear that *AssignCode* performs at most n iterations, and that all the elements of Σ are visited once and only once. Let $\Sigma = \{a_1, \ldots, a_n\}$ be such that its element indices $\{1, \ldots, n\}$ correspond to the traversal order of Σ by the *AssignCode* procedure. Note that this sequence is a topological ordering of Σ with respect to its partial order. Namely, $a_1 = \bot$ and for all indices i and j between 1 and n,

$$i < j \quad \Rightarrow \quad a_j \not\leq a_i \tag{7}$$

Now then, the codes computed by the *AssignCode* procedure are such that $\gamma(a_1) = 1$, and for $i > 1$,

$$\gamma(a_i) \;=\; 2^{i-1} \vee \bigvee_{x \prec a_i} \gamma(x), \tag{8}$$

where $\prec$ means "immediately less than".

If we can establish that the codes computed by the *AssignCode* procedure are such that there is a '1' in the ith bit position of the code word of x (counting from right to left, starting with position 0) if and only if $a_i \leq x$, then we shall have clearly proved that the codes are boolean vector representations of the set of all lower bounds of x. That is, *AssignCode* computes the reflexive and transitive closure of $\succ$. This is precisely what the following theorem states.

Theorem 1 $\forall x \in \Sigma,\ \forall i,\ 1 \leq i \leq n,\ \ 2^{i-1} \wedge \gamma(x) \;=\; 2^{i-1}$ *iff* $a_i \leq x$.

Proof: We proceed by induction on k, the index of $x = a_k$.

This is clearly true for $k = 1$. Indeed, for $x = \bot$, the question is reduced to showing:

$$\forall i,\ 1 \leq i \leq n,\ \ 2^{i-1} \wedge 1 \;=\; 2^{i-1} \ \textit{ iff } \ a_i \leq \bot.$$

and that is obvious.

Let us now assume that this is true up to some index k; namely,

$$1 \leq j \leq k \quad \Rightarrow \quad 2^{i-1} \wedge \gamma(a_j) \;=\; 2^{i-1} \tag{9}$$

Now, by (8),

$$\begin{aligned} 2^{i-1} \wedge \gamma(a_{k+1}) &= 2^{i-1} \wedge (2^k \vee \bigvee_{x \prec a_{k+1}} \gamma(x)) \\ &= (2^{i-1} \wedge 2^k) \vee (2^{i-1} \wedge \bigvee_{x \prec a_{k+1}} \gamma(x)) \\ &= (2^{i-1} \wedge 2^k) \vee \bigvee_{x \prec a_{k+1}} (2^{i-1} \wedge \gamma(x)) \end{aligned}$$

Thus, $2^{i-1} \wedge \gamma(a_{k+1})$ is the binary *or* of two terms. There are two possible cases to consider, $k = i - 1$ or $k \neq i - 1$.

In the first case, we have $2^{i-1} \wedge 2^k = 2^{i-1}$. We also have $a_i = a_{k+1}$, and thus, *a fortiori*, $a_i \leq a_{k+1}$. On the other hand, considering the second term $\bigvee_{x \prec a_i}(2^{i-1} \wedge \gamma(x))$, we notice by (7) that since $x \prec a_i$, the indices j of the $x = a_j$ elements must be such that $j < i$; but then, $k = i - 1$ entails that $j \leq k$. We can therefore use the induction hypothesis (9) together with the fact that $a_i \leq a_{k+1}$ to conclude that $\bigvee_{x \prec a_i}(2^{i-1} \wedge \gamma(x)) = 2^{i-1}$.

Now, if $k \neq i - 1$, then $2^{i-1} \wedge 2^k = 0$. Thus, $2^{i-1} \wedge \gamma(a_{k+1}) = 2^{i-1}$ if and only if

$$\bigvee_{x \prec a_{k+1}} (2^{i-1} \wedge \gamma(x)) = 2^{i-1}$$

By the remark (7) made earlier, it is clear that all these x elements are of index smaller than or equal to k, and each must satisfy the induction hypothesis (9). Therefore,

$$\forall x \prec a_{k+1}, \ \ a_i \leq x$$

which entails $a_i \leq a_{k+1}$.

We have thus showed the *"only if"* direction of the proposition to prove; namely,

$$2^{i-1} \wedge \gamma(a_{k+1}) = 2^{i-1} \ \Rightarrow \ a_i \leq a_{k+1}$$

The reverse direction follows directly by observing that two cases are possible: either $a_i = a_{k+1}$ or $a_i < a_{k+1}$. The first one is reduced to $k = i - 1$, and the same reasoning for this case, in the backward direction as above, works to conclude that $2^{i-1} \wedge \gamma(a_{k+1}) = 2^{i-1}$.

As for the other case, $a_i < a_{k+1}$, let us consider all those elements x such that $x \prec a_{k+1}$. For those, we have either $a_i \leq x$—in which case, by (9), $2^{i-1} \wedge \gamma(x) = 2^{i-1}$—or a_i not related to x—in which case, again by (9), $2^{i-1} \wedge \gamma(x) = 0$. The conclusion follows.□

5 Bottom-Up Encoding

Although the encoding method exposed above can be employed, it uses an unnecessary amount of space for code words. Indeed, each code word is of length exactly $n - 1$, where n is the number of elements of the poset. However, in many cases, these code words do not need to be so long. Indeed, there are many situations where the maximum code word length can be cut to about half of the size. Consider, for example, the 16-element poset of Figure 10 (counting the omitted $\perp$). This tree shaped poset can be encoded by the transitive closure method to yield the following 15-bit long codes shown in Figure 11-a. However, as shown in Figure 11-b, it can easily be seen that only 8-bit long words would suffice in this case (see Section 6.1). A further compaction (more evident on large posets) can result from code modulation, as seen in Figure 11-c (see Section 7).

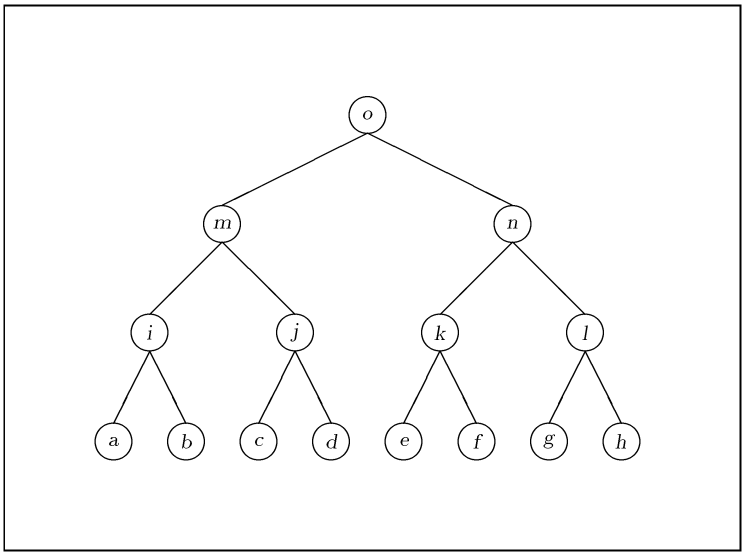


Figure 10: A tree-shaped poset

x	$\gamma(x)$
$\perp$	000000000000000
a	000000000000001
b	000000000000010
c	000000000000100
d	000000000001000
e	000000000010000
f	000000000100000
g	000000001000000
h	000000010000000
i	000000100000011
j	000001000001100
k	000010000110000
l	000100011000000
m	001000000001111
n	010000011110000
o	111111111111111

(a) Transitive closure

x	$\gamma(x)$
$\perp$	00000000
a	00000001
b	00000010
c	00000100
d	00001000
e	00010000
f	00100000
g	01000000
h	10000000
i	00000011
j	00001100
k	00110000
l	11000000
m	00001111
n	11110000
o	11111111

(b) Compact encoding

x	$\gamma(x)$
$\perp$	00 0000
a	01 0001
b	01 0010
c	01 0100
d	01 1000
e	10 0001
f	10 0010
g	10 0100
h	10 1000
i	01 0011
j	01 1100
k	10 0011
l	10 1100
m	01 1111
n	10 1111
o	11 1111

(c) Modulated encoding

Figure 11: Three encodings of the tree poset

In general, it may clearly be necessary to use all $n-1$ bits. A trivial example consists of a flat semi-lattice. However, in practice, tree-like inheritance taxonomies are rather frequent, even if only as parts of posets. Hence, such a saving of space turns out to be substantial in practice.

Let us first see what we can do on the small example of Figure 4. We begin by assigning the code 0 to a. The first layer being $\{a\}$, we obtain the second layer as $\{b, c\}$. Since b has a as unique predecessor, the code of b is computed as $\gamma(a) \vee 2^0 = 0 \vee 1 = 1$. Recording the maximum word length so far as $p = 1$, the same is done to c. Namely, the code of c is obtained as $\gamma(a) \vee 2^1 = 0 \vee 2 = 10$. As before, the next layer obtained from $\{b, c\}$ is $\{d, e\}$. Now, d has more than just one predecessor. Thus, we can assign to d the code $\gamma(b) \vee \gamma(c) = 1 \vee 10 = 11$ without incrementing p since such a code guarantees that, so far, d is strictly greater than all and only its predecessors, and incomparable to all the codes attributed so far to elements which are incomparable to d. Continuing thus, the code assigned to e is $\gamma(c) \vee 2^p = 10 \vee 100 = 110$, bringing p to 3. Finally, the last layer is $\{f\}$, and f has two predecessors. However, this time $\gamma(b) \vee \gamma(e) = 1 \vee 110 = 111$ violates our invariant condition in that it makes it comparable to the code of d although this element is not related to f. Nevertheless, it is possible to reinstate our invariant; indeed, revising the code of d to be the disjunction of what it is (in order to maintain it greater than all its lower bounds) and of 2^p (to guarantee that it remain incomparable to any code so far given to its unrelated elements). Therefore, the final code of d is 1011.

5.1 Method

The idea is that the encoding procedure of Figure 7 is too "generous" in systematically incrementing the word length by 1 each time it assigns a new code. But, in many cases, the code computed as the disjunction of the children's codes would suffice. The only time it is necessary to augment the word length p is when an element has a unique predecessor—in which case it must be made distinguishable from it—or when the disjunctive code so computed is comparable to a code already attributed to an element of Σ known to be incomparable with this element.

5.2 Justification

Informally, if it is ensured that a code is made greater that all and only the codes of its lower bounds, while being incomparable with the codes of incomparable elements, the procedure will be a correct encoding.

This new encoding procedure is obtained from the $AssignCode$ procedure by replacing the call to $EncodeLayer$ by a call to a new procedure, which we name $CompactEncodeLayer$, described in Figures 12, 13, and 14. The $ResolveCodeConflicts$ procedure has for effect to ensure that the invariant condition—imposing that a code attributed to an element be such that it subsumes *all and only* the codes attributed to that element's lower bounds—be satisfied again in case of violation. Clearly, the *all* part is ensured by $IncrementCode$ since it sticks a 1 to the left of the previously assigned code. The *only* part is enforced by the recursive propagation of new such codes to the conflicting node's parents which happened to be already encoded. This is what the $PropagateCode$ procedure does, making sure that all elements known to be incomparable stay so.

The function $IncSet$ is such that $IncSet(x)$ is the set of all elements incomparable with x, element of Σ. These can be pre-computed very efficiently using the classical fast transitive-reflexive closure algorithm (The above-described $\mathcal{O}(n^\alpha \log n)$ version of Warshall-Strassen's method), for the relation $\leq \cup \geq$. $IncSet$ of a_i is the row vector of bits obtained by taking the complement

```
procedure CompactEncodeLayer(L : cochain);
  begin
  for each x ∈ L do
    begin
    let {x_1, ..., x_n} = Children(x) in
      if n > 1
        then γ(x) ← ⋁_{i=1}^{n} γ(x_i)
        else γ(x) ← IncrementCode(x_n);
    ResolveCodeConflicts(x)
    end
  end
```

Figure 12: The CompactEncodeLayer procedure

```
procedure ResolveCodeConflicts(x : element);
  begin
  for each y ∈ IncSet(x) such that Coded(y) do
    case of
      begin
      γ(x) = γ(y) :
        begin
        γ(x) ← IncrementCode(x);
        γ(y) ← PropagateCode(y)
        end;
      γ(x) < γ(y) :
        γ(x) ← IncrementCode(x);
      γ(x) > γ(y) :
        γ(y) ← PropagateCode(y)
      end
  end
```

Figure 13: The ResolveCodeConflicts procedure

```
function IncrementCode(x : element)
  returns : binary;
  begin
  p ← p + 1;
  return 2^(p-1) ∨ γ(x)
  end
```

Figure 14: The IncrementCode instruction

```
procedure PropagateCode(x : element);
   begin
   IncrementCode(x);
   for each y ∈ {z ∈ Parents(x) | Coded(z)} do
      ResolveCodeConflicts(y)
   end
```

Figure 15: The PropagateCode procedure

of the reflexive and transitive closure of "being comparable to" relation ($\leq \cup \geq$), and selecting the i-th row. That is,

$$IncSet(a_i) \quad = \quad [((\leq \cup \geq)^*)^{-1}]_i, \quad i = 1, \ldots, n.$$

where $\Sigma = \{a_1, \ldots, a_n\}$. In addition, this information may be disposed of (garbage collected) before run-time.

6 Variations

6.1 Closed World

There are other sensible choices to make regarding boolean lattice encoding techniques. One of the most obvious is the *closed world assumption*. Under the assumption that new information cannot be added during run time, common in logic programming [6], one can use even shorter binary words for encoding. This encoding algorithm is much like the compact encoding algorithm except that a new bit is only created for minimal types. No new bits are added for types which have unique predecessors, as they are in procedure 12. This encoding corresponds to the set representation of minimal elements in the poset. Thus, any element is the same thing as the set of minimal elements that it subsumes. This closed world encoding (obviously) does not preserve all the information obtainable from the transitive closure encoding, but in some cases this information is unnecessary.

6.2 Disjunctive Objects

We shall now take the freedom of overloading the symbol $\leq$ to mean "less than or equal to" according to the context; *i.e.*, depending on the set of elements it will be used with. The same applies to the symbols for GLB and LUB. This makes presentation lighter.

As remarked earlier in this paper, the reflexive transitive closure code of an element x of Σ is a boolean vector *characteristic* representation of the set of all lower bounds of x. It is characteristic in the very precise sense of a boolean lattice isomorphism between $\mathbf{2}^{\Sigma}$, the powerset of Σ, and $\{0,1\}^{|\Sigma|}$, the set of bit-arrays of length equal to the cardinality of Σ.

Reconsidering the semi-lattice construction of Section 3, one will notice that it is in fact a lattice construction. Indeed, one can define the LUB of two elements in $\mathbf{2}^{(\Sigma)}$as:

$$S \sqcup T \quad = \quad \lceil S \cup T \rceil \tag{10}$$

i.e., the set of maximal elements of the union. It is not difficult to prove that under our assumption of finiteness of Σ, the lattice $\mathbf{2}^{(\Sigma)}$is a complete distributive lattice.

Hence, the coding function γ from Σ to $\{0,1\}^{|\Sigma|}$ may be extended to a function γ_s from $\mathbf{2}^\Sigma$ to $\{0,1\}^{|\Sigma|}$ (thus in particular from $\mathbf{2}^{(\Sigma)}$to $\{0,1\}^{|\Sigma|}$) as:

$$\gamma_s(\{x_1,\ldots,x_n\}) \quad = \quad \bigvee_{i=1}^{n} \gamma(x_i) \tag{11}$$

Note that this function restricted to $\mathbf{2}^{(\Sigma)}$ is always invertible as shown by combining Equations (11) and (5).

When implemented, this coding has the interesting effect of reducing "set-at-a-time" inheritance (unification) from the clearly exponential operation given by Equation (4) to a virtually constant-time *and* operation. In addition, it provides a mild but practical generalization facility which allows induction of LUB symbols which exist explicitly in the poset. For example, if a happens to be the LUB in Σ of $\{a_1,\ldots,a_n\}$, then clearly $\gamma_s^{-1}(\gamma_s(\{a_1,\ldots,a_n\})) = a$.

6.3 Complemented Objects

Let us now consider a simple, albeit quite interesting, extension of the subsumption partial ordering on $\mathbf{2}^{(\Sigma)}$which would be defined on pairs of elements of $\mathbf{2}^{(\Sigma)}$. The idea is to see an object as a pair consisting of a positive object (*i.e.*, everything it can be) and a negative object (*i.e.*, everything it *must not* be). Hence, a general such object may be seen at a set of *examples* and *counter-examples*. Thus, an object of type t is also of type $t_1\backslash t_2$ if and only if $t \leq t1$ and $t \not\leq t2$. Of course, such an object has the same denotation as $\perp$ whenever $t_2 \leq t_1$.

The subsumption ordering is thus extended to complemented objects by:

$$t_1\backslash t_2 \leq t_3\backslash t_4 \;\; \textit{iff} \;\; t_1 \leq t_3 \;\; \textit{and} \;\; t_2 \geq t_4 \tag{12}$$

and thus, the GLB operation for complemented objects is:

$$t_1\backslash t_2 \sqcap t_3\backslash t_4 \quad = \quad (t_1 \sqcap t_3)\backslash(t_2 \sqcup t_4) \tag{13}$$

The encoding function can be extended to complemented objects by:

$$\gamma_c(t_1\backslash t_2) \quad = \quad \gamma_s(t_1) \wedge \overline{\gamma_s(t_2)} \tag{14}$$

Now, decoding complemented codes may still be done by equation 5 with the pragmatic consideration that there is no need to synthesize ever explicitly the negated part of a complemented object. If such a set of counter-examples was needed for some reason, one could always sweep through the entire poset and keep the maximal set of such elements whose codes is not subsumed by the code being decoded.

7 Code Modulation

Although extremely efficient in time for reasonably sized object posets, the encoding techniques presented thus far can become cumbersome in space. This is obvious once one notices that every object must carry a bit for every other object in the poset when using transitive closure, and for each minimal when using the closed world encoding. Encoding posets with hundreds or thousands of minimals on real computers becomes problematic. However, in practice an object taxonomy is often not completely connected. Many applications consist of tree-shaped posets, and in our experience, many object posets consist of several densely connected groups of nodes, with only a few inheritance links into other dense groups. This is natural when one realizes that the groups correspond to semantic groupings—an object called *piston* may have many inheritance links to other *car-parts*, while it may have few links to *musical-instruments*. It is possible to take advantage of this common property of posets to shorten the length of the bit-encodings. This benefit has a small price, in runtime, paid only when elements from incomparable groups participate in disjunctions or negation. Except for some pathological cases, this optimization performs qualitatively better than the transitive closure algorithm in time and space on large posets.

An intuitive notion of our grouping is easily grasped by thinking of a tree of objects. Each of the N leaves is a minimal of the poset, and so is assigned a unique bit. In the algorithm presented so far, each leaf, and each node in the tree must have a bit for at least every leaf, so the total space used just to remember the codes is at least N^2. But if we split the tree in two at the root, and encode each $N/2$ half of the tree separately (using only as many bits for each node as there are leaves in that half of the tree, thus $N^2/4$ bits for each half), and then append a special *group code* to the front of the encoding (append a 01 to the front of each code in one half, and a 10 to the front of each code in the other), and just do the right thing for unification, disjunction, negation, *etc.*, a great space improvement can result (Figure 11-c exhibits a small improvement on a small tree-like poset. Significant improvement can be gained using this technique on large trees.) Now only the number of elements times the length of its *local code* (within the group), plus its group code, or $N(N/2+2)$ bits are required to encode the entire tree, where N^2 were used before. One could further split the tree into smaller subtrees before encoding, adding more grouping bits and reducing the number of nodes in each group. This process can be repeated, but eventually becomes counterproductive as the subtrees become small.

Further, this extension need not be restricted to trees, as long as there are no links into the middle of a group of objects. Splitting a tree in two is easily visualized, but splitting a graph can be more complex. This extension can be seen as simply dividing the original poset vertically and horizontally, while tree splitting only divides posets horizontally. One can draw a picture of a poset on paper, and then draw circles around groups of nodes such that every circle has a unique highest element, a unique lowest element, and the only arcs from elements outside the circle to lower elements inside the circle end on the highest element of the circle, and the only arcs from elements inside the circle to lower elements outside the circle come from the lowest element of the circle. We call the group of elements inside such a circle a *module*. We encode each module of L elements separately, (using L^2 bits), and encode the set of modules M as if the circles on the diagram of the poset were collapsed to points, and encoded as if they form a poset (they do). Each module's code is then appended to the locally determined code of each of its elements. Now the total space used to encode a poset is bounded by the number of objects times the number of groups plus the number of local elements, or $N(M+L)$, where the number of groups M times average number of local elements L must be equal to the total number of objects in the poset ($L \times M = N$).

Yet, there is a possible further abstraction: why stop at groups of elements, why not create groups of groups? If one takes the above described diagram of collapsed modules, and encodes that using modulation, instead of transitive closure, the result is modules of modules. Obviously, this process could be repeated until there is only one module. In the optimistic case, only $N \log N$ bits are needed to encode the entire poset, where the full transitive closure uses N^2. In real posets, this is often impossible, but a close approximation will lead to similarly compact codes.

Theorem 2 *The space used by the above modulated encoding is $\mathcal{O}(N \log N)$.*

Proof: Given a poset of N elements modulated k times, modulation only requires $N(N_1+N_2+\cdots+N_k)$ bits to encode it, where N_x is the number of elements at the x^{th} level, and $N_1 \times N_2 \times \cdots \times N_k = N$. For example, N_k is the number of elements in each group at the bottommost level, or in other words, N_k is the number of elements each circle contained the first time we drew circles. N_x is the number of circles (reduced to points) we drew circles around the $k - x^{th}$ time we drew circles, *etc.* First we show that at any level, equal numbers of elements in each group at that level is optimal. Next we show that $N_1 = N_2 = \cdots = N_k = \sqrt[k]{N}$ is an optimal breakdown of the poset. Finally, we find that the optimal k for a given N is $\log N$.

Given N elements, and M groups, each with the same number of elements, L, $M \times L = N$, we show that moving any number of elements, b from one group to another increases the total required bits. Thus, any configuration where some group has more elements than some other group must not be optimal.

1. $(M+L)$ is the number of bits required to encode each element.
2. $\underbrace{(M+L)+\cdots+(M+L)}_{N}$ is the total number of bits required to encode the poset. If we change it in any way, that change can be seen as a combination of pairwise changes to 2 terms - adding some number of elements to one term, and subtracting the same amount from another. That is,
3. $\underbrace{(M+L+b)+\cdots+(M+L+b)}_{L+b}+\underbrace{(M+L-b)+\cdots+(M+L-b)}_{L-b}+\underbrace{(M+L)+\cdots+(M+L)}_{N-2L}$ which we are trying to show is greater than or equal to the first sum. Subtracting away the $N-2L$ unchanged sums leaves:
4. $\underbrace{(M+L+b)+\cdots+(M+L+b)}_{L+b}+\underbrace{(M+L-b)+\cdots+(M+L-b)}_{L-b} \geq \underbrace{(M+L)+\cdots+(M+L)}_{2L}$. Subtracting $2L$ instances of $(M+L)$ leaves:
5. $\underbrace{(b+b+\cdots+b)}_{L+b}+\underbrace{(-b-b-\cdots-b)}_{L-b} \geq 0$. Rearranging produces:
6. $b(L+b) - b(L-b) \geq 0$.
7. $b = 0$ So completely even modules are best.

Now, we show that each group should contain the same amount of sub-groups as every group on every level. In order for the $N_1 \times N_2 \times \cdots \times N_k = N$ constraint to be met, a total of $N \times K \times \sqrt[k]{N}$ bits must be used. Remember that k is the total number of times we drew circles. (we will show how to decide k later).

Modulation to the same degree at every level, using $N \times k \times \sqrt[k]{N}$ bits, is optimal.

1. $\underbrace{k \times \sqrt[k]{N}+\cdots+k \times \sqrt[k]{N}}_{N}$ is the total number of bits required to encode the poset if modulation is performed to the same

degree at every level. Here we assume that every module at a given level is the same size, as the previous lemma demonstrated was a good idea. If we change the encoding in any way, that change can be seen as a combination of pairwise changes to 2 terms - dividing one term by some positive number, multiplying another term by the same -

2. $\underbrace{k \times \sqrt[k]{N} \times d + \frac{k \times \sqrt[k]{N}}{d} + k \times \sqrt[k]{N} + \cdots k \times \sqrt[k]{N}}_{N}$
   is also a solution. Now we need to show that this sum is greater or equal than the first sum. First, subtract all the unmodified terms from both:
3. $k \times \sqrt[k]{N} \times d + \frac{k \times \sqrt[k]{N}}{d} \geq k \times \sqrt[k]{N} + k \times \sqrt[k]{N}$.
   Multiply both sides by d, and divide by $k \times \sqrt[k]{N}$.
4. $d \times d + 1 \geq d + d$
5. $d^2 - 2d + 1 \geq 0$
6. $(d-1)^2 \geq 0$
7. $d = 1$
   So no change will be beneficial.

So $N \times k \times \sqrt[k]{N}$ is optimal, now we need to find k for a given N - lets take the derivative, set it equal to 0, make sure that that it is a minimum, and solve for k.

1. $\frac{\partial}{\partial k}(N \times k \times \sqrt[k]{N}) = 0$
2. $(N \times \sqrt[k]{N}) + (N \times k \times \frac{\partial}{\partial k}\sqrt[k]{N}) = 0$
3. $(N \times \sqrt[k]{N}) - N \times k \times \frac{\log N \sqrt[k]{N}}{k^2} = 0$.
   Dividing both sides by $(N \times \sqrt[k]{N})$:
4. $1 - k \times \frac{\log N}{k^2} = 0$
   and multiply by k:
5. $k - \log N = 0$
6. $k = \log N$

This turns out to be a minimum, and thus $N \times k \times \sqrt[k]{N}$ becomes $N \times \log N \times \sqrt[\log N]{N}$, which simplifies to $N \times \log N \times e$, which we claim is the minimum total number of bits needed to encode a (perfectly modulatable) poset of N elements using modulation. □

For large posets, a very large improvement in encoded space is expected, and even unreasonably large posets can be realistically coded with this system. For instance, for $N = 100$ this is $100 \times 5 \times e$, or $1,252$, instead of 100^2 or $10,000$, and for $N = 1000$, its $18,778$ instead of $1,000,000$.

Thus far we have been discussing only the space performance of encoding techniques, but the time complexity is also of interest. The GLB operation using transitive closure is $\mathcal{O}(N)$ (or $\mathcal{O}(1)$ on a bit-vector machine), while GLB on modulated codes is $\mathcal{O}(\log N)$ (or $\mathcal{O}(1)$). This is a nice result, since it means that we can have our cake and eat it too - less space and less time. However, these theoretical results are somewhat misleading. On small posets, real computers do behave like bit-vector machines, and so transitive closure and modulation behave approximately the same. On larger posets, real computers do not behave like bit-vector machines, and thus modulation is of most interest for large posets.

For example, for $N = 1000$ space-optimal modulation only requires $18,778$ bits, but uses 18 levels of modulation. But $N = 1000$ could also be broken down into 32 modules of 32 elements (except that 24 of the modules have only 31 elements), requiring $63,256$ bits, but only one level of modulation. If every level of modulation adds potential time cost, the overhead of extra nesting levels may offset the space compactness of optimal groupings.

Also, thus far, we have assumed the poset to be perfectly modulatable. Trees are always perfectly modulatable, as are many sparsely connected posets. But even "messy" object posets can be dealt with by allowing modules of different sizes, even to the extreme of single element modules. This slight generalization allows more modulation than would otherwise be possible on real posets, and although the resulting code sizes won't be optimal, they will be much smaller than if transitive closure had been used.

7.1 Operations on Modulated Codes

This all sounds fine until one reconsiders the above phrase *"just do the right thing for unification, disjunction, negation, etc."* Let us examine what the right thing is.

For the moment we will consider an object poset encoded using only one level of modulation, so each code consists of only one group code, and a local code. The generalization of these operations for modules of modules is straightforward.

First we define $>_m$ to be a function on two codes X and Y, where Xg is X's group code, Xl is X's local code, Yg is Y's group code and Yl is Y's local code, one needs to find the bitwise AND of Xg and Yg, call it Ag, and the bitwise AND of Xl and Yl, call that Al. Within any group, we use -1 as a shorthand for the topmost element's code, and 0 as a shorthand for the bottommost element's code.

$$X >_m Y \text{ iff } \begin{cases} Xg = Yg, \text{ and } Xl > Yl; \\ Xg \neq Yg = Ag; \end{cases}$$

This definition says that X is greater than Y if and only if X and Y are in the same group, and $Xl > Yl$ (using the previously defined $>$ on codes), or X and Y are in different groups, and Y's group is subsumed by X's.

Then, the GLB of X and Y (with $X = \langle Xg, Xl \rangle$, and $Y = \langle Yg, Yl \rangle$, $Ag = Xg \wedge Yg$, and $Al = Xl \wedge Yl$) is:

$$X \wedge Y = \begin{cases} \langle Ag, Al \rangle & \text{if } Xg = Yg = Ag; \\ \langle Xg, Xl \rangle & \text{if } Xg = Ag \neq Yg; \\ \langle Yg, Yl \rangle & \text{if } Yg = Ag \neq Xg; \\ \langle Ag, -1 \rangle & \text{otherwise.} \end{cases}$$

The first possibility is that X and Y are in the same group, and so the result is that groups code appended to the AND of their local codes. The second possibility is that X's group is subsumed by Y's. Thus the result is simply X's original code. The third possibility is that Y's group is subsumed by X's. Then the result is simply Y's original code. The last possibility is that neither group subsumes the other, and thus the result is the topmost element in the group which is the greatest lower bound of the two groups.

Also, to define disjunction of objects, define LUB to compute the bitwise OR of the codes, and call the OR'ed components Og and Ol. Let $\bigtriangledown$ be a binary constructor which represents the modulated disjunction. Thus the LUB of X and Y is:

$$X \vee Y = \begin{cases} \langle Og, Ol\rangle & \text{if } Xg = Yg = Og; \\ \langle Xg, Xl\rangle & \text{if } Xg = Og \neq Yg; \\ \langle Yg, Yl\rangle & \text{if } Yg = Og \neq Xg; \\ \bigtriangledown(X,Y) & \text{otherwise.} \end{cases}$$

The first possibility is that X and Y are in the same group, and so the result is that groups code appended to the OR of their local codes. The second possibility is that X's group subsumes Y's. Thus the result is simply X's original code. The third possibility is that Y's group subsumes X's. Then the result is simply Y's original code. The last possibility, the explicit or, is necessary only in cases where elements of two incomparable groups are ORed. The arguments of the explicit or are the original disjuncts.

One also needs to know how to take GLB's and LUB's of explicit $\bigtriangledown$'s.

$$\bigtriangledown(X,Y) \wedge Z = (X \wedge Z) \vee (Y \wedge Z)$$

$$\bigtriangledown(X,Y) \vee Z = (X \vee Z) \vee (Y \vee Z).$$

Also, define $>_m$ on explicit disjunctions.

$$\bigtriangledown(X,Y) >_m Z \text{ iff } X >_m Z, \text{ or } Y >_m Z$$

$$Z >_m \bigtriangledown(X,Y) \text{ iff } Z >_m X, \text{ and } Z >_m Y$$

Finally, we define negation of objects similar to that in 6.3. Again define complemented objects as pairs of codes, one describing examples, the other describing counter-examples. An object of type t is also of type $t_1\backslash_m t_2$ if and only if $t \leq_m t1$ and $t \not\leq_m t2$. We thus define the $\backslash_m$ or BUTNOT operator as first computing the bitwise AND'ed components Ag and Al, and then

$$X\backslash_m Y = \begin{cases} \langle Xg, Xl\backslash Yl\rangle & \text{if } Xg = Yg = Ag; \\ \bot & \text{if } Xg = Ag \neq Yg; \\ X\backslash_m Y & \text{otherwise.} \end{cases}$$

The first possibility is that X and Y are in the same group. Then $X\backslash_m Y$ is simply $Xl\backslash Yl$ in the common group. The second possibility is that Y's group is subsumed by X's, in which case the result is bottom. Otherwise, the result is an explicit complement of the original arguments.

The obvious generalization of rule 5 to use the modulated encodings and $>_m$ function to decode encoded types is

$$\gamma_m^{-1}(c) \;=\; \lceil\{x \in \Sigma \mid c >_m \gamma_m(x)\}\rceil \qquad (15)$$

In words, $\gamma_m^{-1}(c)$ is the set of maximal elements of Σ whose codes are less than c.

7.2 Implementation of Modulation

Implementation of modulation requires two things - some method to generate the codes, and an efficient implementation of the GLB, LUB, and BUTNOT operations on the resulting codes.

7.2.1 Generating Modules

In order to take advantage of the benefits of modulation, it is necessary to discover the group boundaries, or to draw circles around groups of elements. Below is a sketch of an algorithm to find these modules in an arbitrary poset. This algorithm may not find every module possible, but it does find the vast majority of them, and is able to modulate posets with hundreds of elements in a few seconds. Also, if one wishes to create modules of modules, one simply need invoke the top level function on an already modulated poset.

The basic idea of this algorithm is to group elements together a few at a time until a group grows to some group size bound, in which case that group is not expanded any further, or the set of remaining objects (groups and ungrouped elements) is less than some threshold, in which case the entire process of modulation is complete.

For simplicity in the pidgin code below, a module is named after its first element. Also, it is assumed that the entire poset has been recorded as a set of related pairs, accessible through the previously seen functions $Parents(x)$ and $Children(x)$. It also assumes the operations $Relate(x, y)$ which asserts that $x \prec y$, and $Unrelate(x, y)$ which asserts that $x \not\prec y$ (recall that $x \prec y$ means "x immediately less than y). A few of the procedures used here are not defined, but are obvious from the context.

The algorithm simply looks at each element, attempting to group it together with its parents. If that fails, it tries to group it with a sibling that has exactly the same set of parents as it does. In both types of grouping, it is possible that extra elements (children of the newly added elements) will need to be added to the group in order to satisfy the module requirements. Often, in trying to add the necessary extra children, the group will grow to encompass the entire poset. Thus checks for exceeding the group size bound are added to many of the procedures below. The bound used here is 32, although in fact the bound would depend on many factors such as machine word length *etc.*, and should really be a constant parameter.

The function $Modulate$ (Figure 16) is the main loop of the algorithm. It begins by putting each element of the poset on a queue. It then examines the first thing in the queue. If it is able to grow that element into a larger module, then that new module is recorded, and is pushed on the queue (in order for it to grow further).

$RecordNewModule$ (Figure 17) simply updates the relatedness of elements and modules.

The function $GrowUpward$ (Figure 18) immediately adds all the parents of the element, since an element can never be grouped with only a subset of its parents. It then tries adding extra elements upward, until a homogeneous layer is found. $Homogeneous(s)$ returns true if its argument s is a singleton set, false otherwise.

$GrowSideways$ (Figure 19) is similar to $GrowUpward$, except that it begins by finding its siblings (a cochain), and then finding those siblings which have exactly the same parents as the element. Those full siblings are then added, one at a time, until the bound is reached. As each new sibling is added, some children may have to be added in order to form a group.

$AddNecessaryChildren$ (Figure 20) begins adding children of the new elements until either a group is formed or the module-size limit is reached. In fact, it could be the case that by adding some new parents and some more new children, a group could be formed. However this is unlikely, and complicates the algorithm. Thus a simplification has been made to this function: Whenever one of the new children has any parents, there termed $uncles$, the function returns failure, preventing any such cancerous growth.

function $Modulate(s : set\ of\ elements)$ **returns** $: set\ of\ set\ of\ elements;$
  begin
  $queue \leftarrow s;$
  $modules \leftarrow \emptyset;$
  while $queue \neq \emptyset$ **and** $(|queue \cup modules| > 32)$ **do**
    begin
    $elt \leftarrow$ **pop**$(queue);$
    $newmodule \leftarrow GrowUpward(elt);$
    if $newmodule = \emptyset$ **then** $newmodule \leftarrow GrowSideways(elt);$
    $modules \leftarrow (modules - newmodule) \cup \{newmodule\};$
    $queue \leftarrow (queue - newmodule) \cup \{newmodule\};$
    $RecordNewModule(newmodule)$
    end;
  return $modules$
  end

Figure 16: The main loop of the Modulation algorithm

procedure $RecordNewModule(mod : set\ of\ elements);$
  begin
  $base \leftarrow \bigcup\{Children(x) \mid x \in mod\};$
  $crown \leftarrow \bigcup\{Parents(x) \mid x \in mod\};$
  for each $x \in base$ **do**
    begin
    for each $y \in mod$ **do** $Unrelate(x, y);$
    $Relate(x, mod)$
    end;
  for each $x \in crown$ **do**
    begin
    for each $y \in mod$ **do** $Unrelate(y, x);$
    $Relate(mod, x)$
    end
  end

Figure 17: The RecordNewModule Procedure

```
function GrowUpward(e : element) returns : set of elements;
  begin
  crown ← Parents(e);
  done ← Homogeneous(crown) or (Size(crown) ≥ 32);
  while ¬done do
    begin
    TryAddingMoreAncestors(crown);
    done ← Homogeneous(crown) or (Size(crown) ≥ 32)
    end;
  if Size(crown) ≥ 32 then return ∅
  else return AddNecessaryChildren(crown, e)
  end
```

Figure 18: The GrowUpward Function

```
function GrowSideways(e : element) returns : set of elements;
  begin
  family ← {e};
  crown ← Parents(e);
  siblings ← ⋃{Children(x) | x ∈ crown};
  fullsiblings ← ⋃{x ∈ siblings | Parents(x) = crown};
  oldfamily ← ∅;
  done ← (fullsiblings = ∅) or (|family| > 32);
  while ¬done do
    begin
    brother ← pop(fullsiblings);
    family ← AddNecessaryChildren({brother}, family);
    if |family| ≤ 32 then oldfamily ← family;
    done ← (fullsiblings = ∅) or (|family| > 32);
    end;
  return oldfamily
  end
```

Figure 19: The GrowSideways Function

function $AddNecessaryChildren(added, old : set\ of\ elements)$
  returns $: set\ of\ elements;$
 begin
 $new \leftarrow added - old;$
 $group \leftarrow new \cup old;$
 $newkids \leftarrow \bigcup\{Children(x) \mid x \in new\};$
 $uncles \leftarrow \bigcup\{Parents(x) \mid x \in newkids\};$
 if $Size(group) > 32$ **then return** $\emptyset$
 else
 if $new = \emptyset$ **then return** old
 else
 if $uncles \not\subseteq group$ **then return** $\emptyset$
 else
 return $AddNecessaryChildren(newkids, group)$
 end

Figure 20: The AddNecessaryChildren Function

7.2.2 GLB, LUB, and BUTNOT

In our formulation, GLB is an extremely efficient operation. However, and LUB and BUTNOT are not always as efficient. Uses of the LUB operation on elements from different modules can incur performance penalties in some cases, due to the need for explicit disjunctions. Similarly, BUTNOT of elements from incomparable groups sometimes produces explicit negations. However, in many applications, disjunction and negation are infrequent or nonexistent. In these cases, it is advantageous to modulate the encoding as much as possible. Often the decision to modulate or not to modulate is dependent on details of the system implementation, including hardware architecture (such as machine word-size, microcoding of multi-word bit-vector operations, *etc.*) In particular, it is not advantageous to modulate posets which have fewer elements than the underlying machine word has bits. This is due to the fact that bitwise-and of two bit vectors shorter than the word-size of the machine tends to be the fastest operation possible. Thus on many personal workstations posets with less than 32 elements should be encoded using the transitive closure algorithm, and only larger posets should be broken down.

7.3 Proof of the Pudding

Variants of these algorithm have been implemented in Common Lisp, and have been benchmarked on a Symbolics 3640. The benchmarks were collected by building a series of trees (which are perfectly modulatable), and then adding some number of randomly generated links. Any links which would cause loops or which were redundant (because of transitivity) were ignored. The trees were of exponential nature (the branching factor at depth D was D+1), which generally corresponds to the posets we have encountered in practice. Some timings were collected for fixed-arity trees and for posets from actual practice which correlated well with our timings.[4]

The first set of timings, presented in figure **??** represents the time (in millionths of a second)

[4]Thanks go to Jungyun Seo for "Babel", his library database, and others for smaller sample posets which helped drive the development of this technique.

one GLB takes to compute using transitive closure (the star shaped datapoints), and modulation (the circular datapoints). These times were gathered for the posets that were successfully modulated by our algorithm (see the next set of benchmarks) by randomly selecting 10 pairs of elements, and then finding the minimum time necessary to compute the GLB of each pair, and then dividing by 10. The selection of the minimum time (rather than average) is justified by the multitasking nature of the Symbolics machine. The same procedure was used for timing the Modulated GLB and Transitive Closure GLB.

With fewer than 32 elements in the poset, both GLB operations are exactly one *logand* instruction, which takes between .000004 and .000008 seconds. With more than 32 elements, as can be seen in that diagram, modulated GLB is faster than transitive closure GLB by a factor of at least three. As the size of the poset increases, modulation outperforms transitive closure by larger and larger margins.

However, this performance advantage only exists for certain posets. In fact, only posets which fall in the shaded area Figure **??** are expected to exhibit such performance gains. The dashed vertical line is located at the 32 element mark, where the modulation and transitive closure algorithms diverge. The open circles mark the maximum number of links for which fifty percent of our randomly generated posets were successfully modulated. "Success" is defined to be breaking the poset down into 32 or less modules. Thus below the curved solid line the performance of figure **??** is expected. Below that line and to the right of the dashed vertical line (the shaded area) one can expect at least a factor of three performance gain of modulation over transitive closure. Above that line it is still possible that modulation will succeed, and it is possible that if modulation "fails" it will still outperform transitive closure. However these possibilities are unlikely.

The slanted dotted line represents the effect of tree-splitting, an easy to implement restriction on modulation discussed earlier. As the diagram shows, tree splitting accounts for a large part of the expected gain of modulation. The x's and the curved broken line to the left of the dashed vertical line represent the maximum possible number of links for a given number of elements - any more than 6 links between 5 elements of a poset must be redundant or inconsistent.

These benchmarks reflect both the quality of the implementation of the GLB operations (for both transitive closure and modulated), and the quality of the $Modulate$ routine which finds modules. We believe our implementations to be of high quality, but others may be able to do better. It should also be noted that these benchmarks compare two of our encoding techniques. All of our encoding techniques qualitatively outperform standard methods of implementing inheritance, which can easily be exponential in the size of the poset, where ours are linear or better.

7.4 Modulated Variations

Generalizations of the modulation theme are possible. In fact, it is possible to relax the requirements on modules to allow multiple topmost elements, and multiple bottommost elements in each module. If the upper surface (defined to be all the elements of a group which are immediate descendants of elements not in the group) is upward-homogeneous (defined here to mean that all elements have exactly the same set of parents), and the lower surface is downward-homogeneous, (defined similarly) then the group is a module. As described earlier, modules have singleton upper and lower surfaces, which are trivially homogeneous. In fact, the algorithm presented above is able to find modules which have large surfaces if the below definition of $Inhomogeneous2$ replaces $Inhomogeneous$ in the algorithm. $Inhomogeneous2$ simply tests to see if all the elements

Figure 21:
Average Time to Compute
Modulated and Transitive Closure GLB

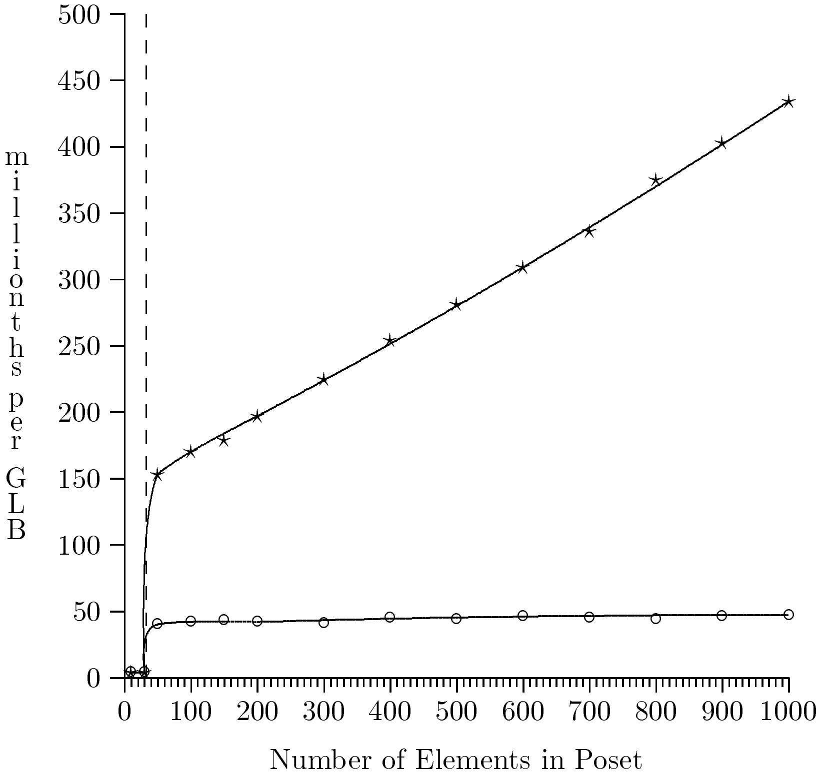
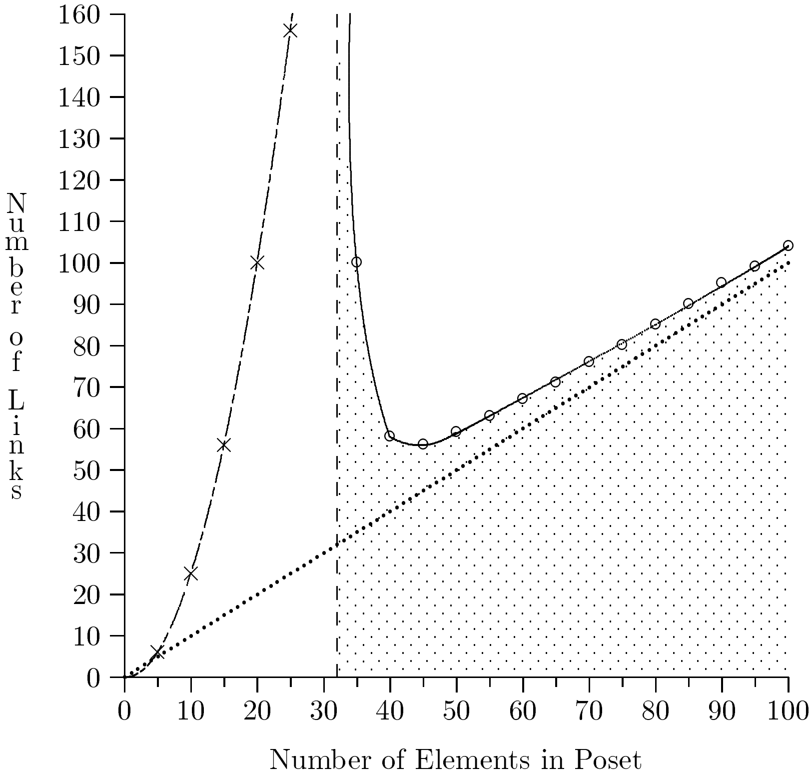


Figure 22:
Posets for which modulated GLB
is faster than transitive closure GLB.



---

function $Inhomogeneous2(crown : set\ of\ elements)$
  returns $: boolean$;
 begin
 $parentset \leftarrow Parents(\mathbf{pop}(crown))$;
 $oksofar \leftarrow true$;
 for each $x \in crown$ **do**
  if$Parents(x) \neq parentset\ oksofar \leftarrow false$;
 return $oksofar$;
 end

---

presented have exactly the same set of parents.

Above, we have assumed that the transitive closure encoding is used to determine the local code, and the group code of modules. It is also possible to use other variations, such as the closed world encoding. However, using alternate local encoding schemes sometimes has subtle effects. Using the closed world encoding, tree splitting is the dominant operation; it is very difficult to divide a closed-world poset vertically into modules. Even more mundane "encoding" schemes, such as depth first search, could be used at one or more levels of a modulated poset. For emphasis on certain aspects of performance, alternate encoding schemes could be used at every level. This mix-and-match approach could be especially useful in very large posets.

Also, dynamic signature changes are much less costly for modulated codes than for the transitive closure generated codes. Using transitive closure encoding, making any changes to the poset after its been encoded are extremely expensive. In fact, if there is a change to the poset at an element E, every element which subsumes E must be recoded. However, using modulated codes, only the elements which subsume E and are in the same module as E must be recoded, unless the change made causes a link between modules, in which case many more must be recoded. Often the overhead of changing even a small number of codes at runtime prevents taking advantage of this feature of modulated codes, but in some applications dynamic changes to the poset can be facilitated with the use of modules.

Again, the largest consequence of modulating the encoding of a poset is to reduce the space required to store the codes from $\mathcal{O}(N^2)$ to $\mathcal{O}(N \log N)$. But it should also be emphasized that the theoretical complexity of the modulated GLB operation is $\mathcal{O}(\log N)$.

References

[1] Aho, V.A., Hopcroft, J.E., and Ullman, J.D. *The Design and Analysis of Computer Algorithms*. Addison-Wesley (1974).

[2] Albano, A., Giannotti, F., Orsini, R., and Pedreschi, D. The type system of Galileo. In Atkinson, M., Buneman, P., and Morrison, R., editors, *Data Types and Persistence*, Springer-Verlag, Berlin, West-Germany (1988) pp. 101–120.

[3] Aït-Kaci, H. *A Lattice-Theoretic Approach to Computation Based on a Calculus of Partially-Ordered Type Structures*. PhD thesis, Computer and Information Science, University of Pennsylvania, Philadelphia, PA (1984).

[4] Aït-Kaci, H. An algebraic-semantic approach to the effective resolution of type equations. *Theoretical Computer Science*, 45 (1986) pp. 185–215.

[5] Aït-Kaci, H. and Lincoln, P. *LIFE: A Natural Language for Natural Language*. Technical Report ACA-ST-074-88, Microelectronics and Computer Technology Corporation, Austin, TX (February 1988).

[6] Aït-Kaci, H. and Nasr, R. LOGIN: a logic programming language with built-in inheritance. *Journal of Logic Programming*, 3 (1986) pp. 185–215.

[7] Aït-Kaci, H., Boyer, R., Lincoln, P., and Nasr, R. *Efficient Implementation of Object Inheritance*. Technical Report AI-102-87, Microelectronics and Computer Technology Corporation, Austin, TX (1987).

[8] Birkhoff, G. *Lattice Theory*. Volume 25 of *Colloquium Publications*, American Mathematical Society, Providence, RI, third edition (1979).

[9] Bobrow, D. *et al.* Commonloops. In *Proceedings of IJCAI-85*, Los Angeles, CA (1985).

[10] Cardelli, L. *Amber*. Technical Memorandum 11271-840924-10TM, AT&T Bell Labs, Murray Hill, NJ (1984).

[11] Goguen, J. and Meseguer, J. Extensions and foundations of object-oriented programming. *ACM SIGPLAN Notices*, 21:10 (October 1986) pp. 153–162.

[12] Goldberg, A. and Robson, D. *SmallTalk-80: The Language and its Implementation*. Addison-Wesley (1980).

[13] Jaffar, J. and Lassez, J.-L. Constraint logic programming. In *Proceedings of the 14th ACM POPL Symposium*, Munich, West-Germany (January 1987) pp. 111–119.

[14] McAllester, D. Boolean class expressions. *ACM SIGPLAN Notices*, 21:11 (November 1986) pp. 417–423.

[15] Mukai, K. Anadic tuples in prolog. Paper presented at the workshop organized by J. Minker on Foundations of Deductive Databases and Logic Programming, Washington, DC (August 1986).

[16] Parker, D.S. *Partial Order Programming*. Technical Report CSD-870067, Computer Science Department, UCLA, Los Angeles, CA (December 1987).

[17] Smolka, G., Nutt, W., Goguen, J., and Meseguer, J. Order-sorted equational computation. In Aït-Kaci, H. and Nivat, M., editors, *Resolution of Equations in Algebraic Structures*, Academic-Press, Cambridge, MA (Forthcoming).

[18] Stickel, M. Automatic deduction by theory resolution. In *Proceedings of IJCAI-85*, Los Angeles, CA (1985) pp. 1181–1186.

[19] Walther, C. A mechanical solution of Schubert's steamroller by many-sorted resolution. *Artificial Intelligence*, 26 (1985) pp. 217–224.

[20] Weinreb, D. and Moon, D. *Lisp Machine Manual*. Massachussets Institute of Technology, Cambridge, MA (1981).