Sets as Anti-Chains

Carl A. Gunter
University of Pennsylvania

Teow-Hin Ngair
National University of Singapore

Devika Subramanian*
Rice University

Abstract

In this paper we present a theory of anti-chains as data representations for sets in circumstances where sets of interest satisfy properties such as being upward-closed or convex relative to a partial ordering. Our goal is to provide an interface for supplying an implementation of the necessary primitives in a reusable manner and a theory that will facilitate algebraic reasoning about correctness. We present an algebra of anti-chains and illustrate its use in programming and reasoning about a machine learning algorithm.

1 Introduction

A crude way to represent a set is to maintain a list of its elements. Given an ordering for set elements, this can be optimized by maintaining the elements of a set in a structure like a balanced tree. In special circumstances a set can be maintained more indirectly as a predicate that tests set membership. This has the advantage of greater flexibility (such as the ability to represent infinite sets of elements), but it may be so general that it is impossible to implement basic operations efficiently. In this paper we analyze the primitive operations for a representation that can compromise between these two approaches when the sets being represented are known to have certain order-theoretic closure properties.

To begin the discussion with an illustrative example, suppose we must maintain sets of strings of digits, supporting operations like testing whether a particular string is in a set and binary operations like taking the union or intersection of two such sets. We can order digit strings by the prefix order (for example, 01 is a prefix of 012 and 013 but not of 001) and represent them as balanced trees of strings, but it may become costly to maintain large sets in this way. We could introduce a logic capable of expressing properties of strings and then use predicates to represent sets; the efficiency of this approach will depend on the kinds of predicates we expect to use and how expensive it will be to test them.

*Email addresses: gunter@cis.upenn.edu, teowhin@iss.nus.sg, devika@cs.rice.edu.

However, there are circumstances where something like a list of elements can be used (even to represent infinite sets), but where it is not essential to include all elements in order to represent the whole set. Suppose we know a special fact about the sets of strings that interest us: if $s \in S$ for one of the sets S, and s is a prefix of s', then $s' \in S$ too. In this case we do not need to maintain a tree of *all* of the elements in S because the presence of some can be inferred from that of others. For instance, if $01 \in S$, then 012 and 013 are also in S. In particular, we can represent S as a set of strings S' having the property that no two strings in S' are prefixes of one another, and every element of S has an element of S' as one of its prefixes. This provides a compact representation for sets which are infinite, although only sets that have finitely many distinct prefixes could be represented in this way. To determine of a string s whether it is in S, one simply checks whether it has any of the elements of S' as a prefix.

Even if checking membership is no problem, the representation will be useless if it is not possible to carry out other basic operations with it. For example, given sets S and T represented by prefixes from S' and T', how does one represent a set like $S \cup T$? This could be done by testing membership in S or T, but we can represent this test by using $S' \cup T'$. This can be optimized by removing elements $u \in S' \cup T'$ if there is an element $u' \in S' \cup T'$ such that u' is a proper prefix of u. The situation is a little more complicated for the intersection operation, but $S \cap T$ can also be calculated in terms of S', T'.

This paper aims to provide a collection of primitives for computing with sets using this optimization. We have two primary goals. First, we want to provide an order-theoretic and algebraic basis for reasoning about the representation. Second, we wish to isolate the primitives on which it is based and show how they can be used to formulate an interface to applications which can make use of the representation. To these ends, the paper is organized as follows. We begin with a discussion of anti-chains and introduce the anti-chain operations. We then illustrate how these operations can be used to describe a well-known machine learning algorithm which depends on an anti-chain representation and how our semantic theory can be used to prove its correctness. Following this illustration we show how the anti-chain algebra can be used to design an interface to reusuable implementations based on the parameterized module system of the Standard Meta-Language (SML). An SML signature for anti-chains appears at the end of the paper.

This is an abstract from a full paper on anti-chains as a data representation. Readers are referred to the full paper for examples, proofs of results, and further applications. The topic here sketches the anti-chain interface and its semantics. A related paper [2] focuses on AI applications of the order-theoretic formulation of partial information, but is missing the insight provided by recognizing anti-chains as the key underlying structure. In the present paper we make the overly-strong simplifying assumption that we are dealing with finite lattices. The full paper provides a thorough treatment of the more general cases.

2 Anti-Chains

We begin by describing definitions and basic results about partial orders and anti-chains. A *poset* is a set P together with a binary relation $\preceq$ that is reflexive ($x \preceq x$), antisymmetric ($x \preceq y$ and $y \preceq x$ implies $x = y$), and transitive ($x \preceq y$ and $y \preceq z$ implies $x \preceq z$). A set $S \subseteq P$ is said to be *downward closed* or *lower* if $x \in S$ and $y \preceq x$ implies that $y \in S$. Given a set $S \subseteq P$, there is a smallest downward-closed subset of P that contains S which is denoted by $\downarrow S = \{x \in P \mid x \preceq y \text{ for some } y \in S\}$. It is easy to see that S is downward-closed if, and only if, $S = \downarrow S$. Dually, S is said to be *upward closed* or *upper* if $S = \uparrow S$ where we define $\uparrow S = \{x \in P \mid y \preceq x \text{ for some } y \in S\}$. It will save us some extra parentheses later if we assume that the unary operations of downward and upward closure bind more strongly than various set-theoretic operations. For example, $\downarrow S \cap \downarrow T$ is the same as $(\downarrow S) \cap (\downarrow T)$.

Let us say that an element $x \in S \subseteq P$ is *maximal* in S if, for every $y \in S$, $x \preceq y$ implies $y = x$. It is said to be *minimal* in S if $y \in S$ and $y \preceq x$ implies $y = x$. Let us denote by $\max(S)$ and $\min(S)$ the respective sets of maximal and minimal elements of S. In certain circumstances, lower and upper sets can be represented by their upper and lower boundaries. For instance, if P is a finite poset and S is a lower subset of P, then $S = \downarrow \max(S)$. Similarly, if S is an upper set, then $S = \uparrow \min(S)$. Even when a poset is infinite, it may be possible to use boundaries to represent subsets if the sets are 'finitely-generated'. These facts are special instances of the following:

Lemma 1 *Let P be a poset and suppose S' is a finite subset of P. Then $\downarrow S' = \downarrow \max(S')$ and $\uparrow S' = \uparrow \min(S)$.* □

What kinds of subsets of a poset can be the boundaries of its upper and lower subsets?

Definition: Let $P, \preceq$ be a poset. A subset $S \subseteq P$ is an *anti-chain* if it contains no comparable pair of distinct elements, that is, if $x, y \in S$ and $x \preceq y$, then $x = y$. We use the notation $\text{Anti}(P)$ for the set of anti-chains over P. □

If P is a finite poset then the mapping $S \mapsto \uparrow S$ is a bijection between anti-chains and upward-closed subsets of P, while the mapping $S \mapsto \downarrow S$ is a bijection between anti-chains and downward-closed subsets. The functions min and max are inverses for upward closure and downward closure respectively. The relationship between anti-chains and lower or upper subsets is more complex when the poset is infinite.

Aside from upper and lower sets, another kind of subset that can sometimes be described with anti-chains is a convex set:

Definition: Let $P, \preceq$ be a poset. A subset $C \subseteq P$ is said to be a *convex set* if, for each $x, y, z \in P$, the conditions $x \preceq y \preceq z$ and $x, z \in C$ imply that $y \in C$. □

This is the order-theoretic analog of convexity in the plane, where a region C is defined to be convex if the elements on a line between any two points in C are

also contained in C. Convex sets can be described in a variety of ways. Let P be a poset and suppose that U_1, U_2 are upper sets and L_1, L_2 are lower sets of P. Each of the following subsets of P is a convex set: $U_1 \cap L_1$, $U_1 - L_1$, $L_1 - U_1$, $L_1 - L_2$, $U_1 - U_2$. In cases where upper and lower sets can be represented by anti-chains, it follows that convex sets can be represented by pairs of anti-chains. For a finite poset $P, \preceq$ and convex set C,

$$C = \{y \in P \mid \exists x \in \min(C).\ \exists z \in \max(C).\ x \preceq y \preceq z\}.$$

In other words, a convex set in a finite poset can be represented by its sets of maximal and minimal elements—a pair of anti-chains. The result is not true of posets in general, however. For instance, the collection of rational numbers q such that $0 < q < 1$ is a convex set, but it has no maximal or minimal element. Treating the infinite case involves other conditions which are beyond the scope of this paper.

3 Operations on Anti-Chains

Our goal is to represent operations that we would like to perform on upward-closed and downward-closed sets indirectly in terms of operations on anti-chains. Let $S - T$ be the set of elements in S that are not in T. Aside from testing set membership, here are operations that will interest us:

Difference: $\uparrow(S - T)$ where S and T are both upper sets and $\downarrow(S - T)$ where S and T are both lower sets.

Union: $S \cup T$ where S and T are both upper sets or both lower sets.

Heterogeneous intersection: $\downarrow(U \cap L)$ and $\uparrow(U \cap L)$ where U is an upper set and L is a lower set.

Homogeneous intersection: $S \cap T$ where S and T are both upper sets or both lower sets.

A few notes on the form of these operations may clarify some apparent lack of uniformity. It is easy to check that the union and intersection of a pair of upper sets is again an upper set. A similar preservation property holds for lower sets. However, $S - T$ may not be an upper set even if S and T are, so it is essential to modify the upward-set difference operation by taking the upward closure $\uparrow(S - T)$ of their ordinary set-theoretic difference. A similar consideration holds for heterogeneous intersections.

Let us begin with the operation $\downarrow(S - T)$ where S and T are lower sets represented by anti-chains S' and T' where $S = \downarrow S'$ and $T = \downarrow T'$. We want the anti-chain R' such that $\downarrow R' = \downarrow(S - T)$. This set R' can be shown to be the set of those elements $x \in S'$ such that there is no $y \in T'$ such that $x \preceq y$. This collection is easy to calculate: one simply takes each element of T' in turn and removes all of the elements of S' that it dominates—when all of the elements of

T' have been treated in this way, we are done. Now, we want to describe this as a binary operation on anti-chains. It will be helpful to remember that this operation is intended to represent the downward closure of a difference operation but is not itself the downward closure of a difference, so we need to denote it with a different symbol. We therefore write

$$\boxed{S' -^l T' = \{x \in S' \mid \forall y \in T'.\ x \not\preceq y\}}$$

where the superscript l is intended as a reminder that lower sets are being manipulated (via their representation as anti-chains). The desired property is: $\downarrow(S' -^l T') = \downarrow(\downarrow S' - \downarrow T')$. It is also possible to show that if we define

$$\boxed{S' -^u T' = \{x \in S' \mid \forall y \in T'.\ x \not\succeq y\}}$$

where we write $x \succeq y$ to mean that $y \preceq x$, then $\uparrow(S' -^u T') = \uparrow(\uparrow S' - \uparrow T')$.

The union of two sets is easy to represent in these terms. If S', T' are anti-chains, then $\downarrow S' \cup \downarrow T' = \downarrow(S' \cup T')$. Unfortunately, $S' \cup T'$ may not be an anti-chain, so it is essential to take a max. In the upper case it is necessary to take a min:

$$\boxed{\begin{array}{l} S' \cup^l T' = \max(S' \cup T') \\ S' \cup^u T' = \min(S' \cup T') \end{array}}$$

We have $\downarrow(S' \cup^l T') = \downarrow S' \cup \downarrow T'$ and $\uparrow(S' \cup^u T') = \uparrow S' \cup \uparrow T'$. Now, if U is an upper set and L is a lower set, then

$$\boxed{U' *^l L' = \{x \in L' \mid \exists y \in U'.\ y \preceq x\}}$$

and $\downarrow(U' *^l L') = \downarrow(\uparrow U' \cap \downarrow L')$. We deliberately avoid using the intersection symbol $\cap$ here for heterogeneous intersection because it will be used for homogeneous intersection. The upper heterogeneous intersection has a similar representation:

$$\boxed{U' *^u L' = \{y \in U' \mid \exists x \in L'.\ y \preceq x\}}$$

which satisfies $\uparrow(U' *^u L') = \uparrow(\uparrow U' \cap \downarrow L')$.

Of the eight basic operations we set out to describe, we have now covered six: lower and upper difference ($-^l$ and $-^u$), lower and upper union ($\cup^l$ and $\cup^r$), and the heterogeneous lower and upper intersection ($*^l$ and $*^u$). This leaves the two most difficult and most interesting operations: homogeneous lower and upper intersection. Let us focus on homogeneous upper intersection; the issues with homogeneous lower intersection will be dual. Suppose we want to compute the intersection of upper sets S and T from their representations as anti-chains S' and T' where $S = \uparrow S'$ and $T = \uparrow T'$. Taking the intersection $S' \cap T'$ is clearly incorrect. To see why, consider a poset P with three elements $\{a, b, c\}$ where the only order relationships are $b \preceq a$ and $c \preceq a$. If $S = \{a, b\}$ and $T = \{a, c\}$, then $S' = \{b\}$ and $T' = \{c\}$. While $S \cap T = \{a\}$, we have $S' \cap T' = \emptyset$. In this

case, the value of $S' \cap^u T'$ clearly needs to be $\{a\}$ rather than $\emptyset$. The question, therefore, is how this is calculated.

A poset $P, \preceq$ is said to be a *lattice* if each of its finite subsets has a least upper bound and a greatest lower bound. Given $x, y \in P$, let us write $x \wedge y$ for the least upper bound and $x \vee y$ for the greatest lower bound of x and y. The least upper bound of the empty set is the least element of the lattice and we denote it by $\bot$; the greatest upper bound of the empty set is the greatest element of the lattice and it is denoted by $\top$. Basic properties of lattices can be found in a source like [1].

Let us now consider how to calculate intersections of lower and upper subsets of lattices in terms of anti-chains. Let P be a lattice. For anti-chains S', T' of P, define

$$\boxed{\begin{array}{l} S' \cap^l T' = \max\{x \wedge y \mid x \in S' \text{ and } y \in T'\} \\ S' \cap^u T' = \min\{x \vee y \mid x \in S' \text{ and } y \in T'\} \end{array}} \quad \text{(for lattices)}$$

It is possible to show that $\downarrow(S' \cap^l T') = \downarrow S' \cap \downarrow T'$ and $\uparrow(S' \cap^u T') = \uparrow S' \cap \uparrow T'$.

4 Version Spaces

Let us now consider how the anti-chain operations we have described are related to the *version space* algorithm of Mitchell [6]. The results build on ideas of Hirsh [3] and Mellish [4], which generalize Mitchell's original construction, by exploring the role of the anti-chain operations described in the previous section.

For our purposes a *concept space* is a set of sets P such that $\emptyset \in P$ and $UP \in P$ where $UP = \{a \mid a \in x \text{ for some } x \in P\}$. The elements of UP are called *instances* and the elements of P are called *concepts.* A concept space is partially ordered by set inclusion, that is, $x \preceq y$ iff $x \subseteq y$. It is important to note that nothing in general is known about the structure of this poset; in particular, P will not typically be the collection of all subsets of UP (this deviation being the 'representational bias' [5] of the concept space). If $x \preceq y$ then we say that x is *more specific* than y or we say that y is *more general* than x. A *training set*[1] over a version space P is a pair (Γ, Δ) where $\Gamma \subseteq UP$ is called the set of *positive instances* and $\Delta \subseteq UP$ is called the set of *negative instances.* The *version space* $\mathcal{K}(\Gamma, \Delta)$ determined by (Γ, Δ) is defined by the equation:

$$\mathcal{K}(\Gamma, \Delta) = \{x \in P \mid \Gamma \subseteq x \subseteq \overline{\Delta}\}$$

where $\overline{\Delta}$ is the complement of Δ in UP. This collection represents the set of concepts consistent with the training set (Γ, Δ). Computationally, the goal is to calculate new version spaces as the training set is extended. In [6] this was done by an algorithm for calculating $\mathcal{K}(\Gamma \cup \{a\}, \Delta)$ and $\mathcal{K}(\Gamma, \Delta \cup \{a\})$ from $\mathcal{K}(\Gamma, \Delta)$ for any instance a. This idea was refined by Hirsh [3] to the question of how one

[1] This is a slight misnomer because a training 'set' is actually a pair of sets in this formulation.

efficiently calculates $\mathcal{K}(\Gamma_1 \cup \Gamma_2, \Delta_1 \cup \Delta_2)$ in terms of $\mathcal{K}(\Gamma_1, \Delta_1)$ and $\mathcal{K}(\Gamma_2, \Delta_2)$. He aptly terms the solution for this, which can be viewed as a generalization of Mitchell's original approach, the *incremental version space merging* algorithm. Our goal is to show how this algorithm can be understood directly in terms of the anti-chain operations.

The key observation concerning the representation of version spaces is that the version space induced by any training set is a *convex set* and may therefore be represented in one of the ways discussed earlier. In particular, [6] represents a version space as a pair of anti-chains consisting of its maximal and minimal elements. The incremental merging algorithm is defined in terms of this succinct data representation.

Lemma 2 *If P is a concept space and (Γ, Δ) is a training set, then the version space $\mathcal{K}(\Gamma, \Delta)$ is a convex set.* □

This tells us that the version spaces over a finite concept space P can be represented by a pair consisting of their maximal and minimal elements. Another way to view this, in light of the correspondence between anti-chains and upper or lower sets, is to view a convex set C as a pair consisting of an upper set, $\uparrow S$, and a lower set, $\downarrow G$, where $S = \min(C)$ is the set of most specific elements of C and $G = \max(C)$ is the set of its most general elements.

The key question is how to compute the desired operations on version spaces in terms of these anti-chains. To do this, we first note that we have the following equation for training sets (Γ_1, Δ_1) and (Γ_2, Δ_2):

$$\mathcal{K}(\Gamma_1 \cup \Gamma_2, \Delta_1 \cup \Delta_2) = \mathcal{K}(\Gamma_1, \Delta_1) \cap \mathcal{K}(\Gamma_2, \Delta_2). \tag{1}$$

This means that it suffices to be able to compute the intersection of version spaces *in terms of the pairs of anti-chains that represent them.* We can describe how to do this quite succinctly in terms of our collection of anti-chain operations by the following definition:

$$\boxed{(S_1, G_1) \cap^c (S_2, G_2) = ((S_1 \cap^u S_2) *^u (G_1 \cap^l G_2), (S_1 \cap^u S_2) *^l (G_1 \cap^l G_2))} \tag{2}$$

Correctness of the equation is described by the following:

Theorem 3 *Let P be a finite concept space that is a lattice and suppose (Γ_1, Δ_1) and (Γ_2, Δ_2) are training sets with*

$$\begin{array}{ll} S_1 = \min(\mathcal{K}(\Gamma_1, \Delta_1)) & G_1 = \max(\mathcal{K}(\Gamma_1, \Delta_1)) \\ S_2 = \min(\mathcal{K}(\Gamma_2, \Delta_2)) & G_2 = \max(\mathcal{K}(\Gamma_2, \Delta_2)). \end{array}$$

If $(S_3, G_3) = (S_1, G_1) \cap^c (S_2, G_2)$ and $C = \mathcal{K}(\Gamma_1 \cup \Gamma_2, \Delta_1 \cup \Delta_2)$, then $\min(C) = S_3$ and $\max(C) = G_3$. □

From a computational perspective, there is some redundancy in Equation (2) since the anti-chains $S = S_1 \cap^u S_2$ and $G = G_1 \cap^l G_2$ are apparently calculated twice each. Moreover, there is an optimization one can make in calculating

Table 1: Incremental Version Space Merging Algorithm.

```
function mergeVS((S1, G1), (S2, G2)) =
  let value U = S1 ∩ᵘ S2
      and   L = G1 ∩ˡ G2
      and   S3 = U *ᵘ L
      and   G3 = S3 *ˡ L
  in  (S3, G3)
  endlet
```

the second component if one is given the value of the first. A more realistic algorithmic presentation of version space merging is given in Table 1 where the program is described in pseudo-code using the appropriate four anti-chain operations. In the code there, the `let` declaration binds each variable in the environment obtained from the equations before it, that is, binding is sequential (rather than simultaneous).

The calculation in Table 1 is essentially the same as those in [6, 3]. It differs from the value described in Equation (2) in calculating $S_1 \cap^u S_2$ and $G_1 \cap^l G_2$ only once, of course, but also by using S_3 to compute G_3 rather than using $U = S_1 \cap^u S_2$ for this purpose. To see that it is equivalent to the value described by Equation (2), we need only establish the following equation

$$(U *^u L) *^l L = U *^l L. \tag{3}$$

for anti-chains U and L. To see that this implies that the value `(S3, G3)` computed in Table 1 matches the value calculated in Equation (2), just let $U = S_1 \cap^u S_2$ and $L = G_1 \cap^l G_2$ and substitute using Equation (3). To see why (3) holds, let us analyze what it means to have $x \in (U *^u L) *^l L$. Unfolding the definitions of upper and lower heterogeneous intersection yields the assertion: $x \in L$ and $\exists y \in (U *^u L).\ y \preceq x$. This just means:

$$x \in L \text{ and } \exists y \in U.\ (\exists x_0 \in L.\ y \preceq x_0) \text{ and } y \preceq x.$$

But this is clearly equivalent to: $x \in L$ and $\exists y \in U.\ y \preceq x$. This is the same as $x \in U *^l L$. We note that the above constitutes a rigorous proof of correctness of a standard optimization used in implementations of the version space algorithm[6, 3]. The abstract description of the algorithm in terms of antichains and operations on them, enables us to easily establish proofs of correctness for algorithmic optimizations.

The version space learning algorithm itself proceeds by repeated merging of version spaces. The algorithm assumes that we are given a way to obtain new information in the form of a version space; this may be done by extending a training set by examining a new instance, but any method—including the exploitation of domain knowledge—would suit the algorithm. The new information is merged with the old until the desired level of accuracy is achieved.

Unfortunately we lack the space here to treat this topic more fully. To be significant, an anti-chains implementation to support version spaces must treat a somewhat broader class of posets than finite lattices. The full paper includes a thorough treatment of the needed generalization.

5 Interfaces

We now describe interfaces for an SML implementation of anti-chains over lattices. These interfaces were developed in the implementation of an AI algorithm that operates on anti-chains—the assumption-based truth maintenance system, to be precise—but is equally usable for the version space algorithm. An SML *signature* is a list of names expected to be present in an implementation of the signature; such implementations are called *structures.* Signatures contain the names of structures, types, exceptions, and values. In a signature, a value name is given together with its type. For example, to declare that the name `singleton` denotes a value mapping `elt`'s to `ac`'s, one includes the line

```
val singleton : elt -> ac
```

in the signature. The signature for anti-chains is given at the end of the paper. An anti-chain structure defines a type called `ac` for anti-chains and a type called `elt` for the elements of anti-chains. Anti-chains are special kinds of subsets of a poset, so it makes no sense simply to speak of anti-chains independently of the posets over which anti-chains are being taken. An `ANTICHAIN` implementation is (typically) obtained by applying a *functor* (SML parameterized module) to a structure implementing a poset. For instance, the signature `LATTICE` in Table 2 provides the necessary operations to define a functor

```
functor Lattice2AC : LATTICE -> ANTICHAIN
```

where the signature `LATTICE` is given in Table 2.

A breif explanation of `LATTICE` is in order. Its semantics is given by the mathematical lattice axioms together with the stipulation that `sortOrd` is a linear order. The values of `relationship` represent $\prec$ (`Less`), $\succ$ (`Greater`), and $=$ (`Equal`). In a lattice, a given pair of elements may satisfy none of these relationships, so the `latOrd` operation takes a pair of lattice elements and produces a `relationship` *option* as its output. An element of `relationship option` either has the form `Some x` where `x` is a `relationship` or has the form `None`. The function `sortOrd` provides support for the representation of sets of lattice elements as trees. It is important to appreciate that the ordering used for trees generally *must* be different from the ordering `latOrd` on the lattice since a lattice need not be a linear ordering. Note in particular that if we are representing anti-chains relative to the lattice ordering then there will be *no* relationship between pairs of elements of the anti-chain!

Now let us turn to the signature `ANTICHAIN` which appears at the end of the paper. The semantics of most of the interface operations in the last half of the signature are described by the mathematics in Section 3 (assuming a self-evident

Table 2: LATTICE

```
signature LATTICE =
  sig
    type elt
    datatype relationship = Less | Greater | Equal
    datatype 'a option = Some of 'a | None
    val latOrd : elt * elt -> relationship option
    val bottom : elt
    val meet : elt * elt -> elt
    val top : elt
    val join : elt * elt -> elt
    val sortOrd : elt * elt -> relationship
  end (* LATTICE *)
```

mapping of the names). Constants and operations in the first half are taken by analogy with other operations in the sets signature of the SML/NJ library (version 0.93). The semantics of these are described succinctly in comments. The important thing to note is whether an operation refers to the anti-chain or to the downward- or upward-closed set that the anti-chain is meant to represent. This distinction means nothing for functions like **singleton** and **equal** which are the same regardless of which meaning is taken. However, it is essential to note that the function **numElts** gives the number of elements in the representing anti-chain rather than the number of elements in a lower or upper set represented by it. For the functions **app**, **revapp**, **fold**, and **revfold**, the order that is *increasing* or *decreasing* must, of course, be **sortOrd**.

The key points about these interfaces and the way they have been described are a familiar collection of conditions for reusable code. First, the semantics of the interfaces are given abstractly so that the mathematical model is clear and does not over-constrain the implementation. Second, the choice of what to include in the interface was based on a selection of the mathematical primitives needed to express the algorithms which the implementations of the interfaces are intended to support. Third, and finally, the interface language in which the sets of operations are described provides types and abstractions supporting a substantial but computationally feasible part of the task.

Although an emphasis on mathematical and implementation independent descriptions is desirable, the choice of interfaces will be significantly influenced by a tension between available implementation techniques and the kind of reuse that the programmer is trying to achieve. The interfaces provide a vocabulary in which to discuss these trade-offs more formally. Let us illustrate. In some contexts using **Lattice2AC** may prove awkward or inefficient. For example, although the mathematics of lattices calls for top and bottom elements, it is possible to implement **ANTICHAIN** without using them. Moreover, in some cases where one has a lattice mathematically, one or the other of these elements may

be difficult to implement. Hence it is often desirable to use a 'thinner' lattice signature, `LATTICE'` that omits `top` and `bottom`. In SML an implementation of `LATTICE'` is still an implementation of `LATTICE`, so little is lost by this thinning.

Another serious issue arises when one knows something about the input lattice that can be useful in the efficient implementation of anti-chains over it. For instance, if one knows that the lattice will be a boolean lattice over a finite set of atoms, then the anti-chain implementation may optimized by taking advantage of this fact. A functor from `LATTICE` or `LATTICE'` to `ANTICHAIN` cannot do this because its input interface lacks the needed primitives. Moreover, it is quite simple to describe boolean lattices because all one needs to know are the atoms; the lattice operations can all be defined in terms of whatever representation of sets of atoms one chooses to use. For these two reasons it probably makes more sense to organize code into a functor that takes an 'atoms' model as its input. So, given a signature `ATOMS`, one implements `functor Atoms2AC : ATOMS -> ANTICHAIN`. Whether anti-chains over a lattice are produced using `Lattice2AC` or `Atoms2AC`, the mathematical semantics of the anti-chain operations should remain the same. The implementations will undoubtably differ.

Conclusions and Acknowledgements

We have presented a collection of operations on anti-chains and illustrated their use in describing and reasoning about an algorithm that uses an anti-chain representation as an optimization. We have analyzed some of the issues about how these operations could be provided to applications through reusable implementations. The key to this is the discovery of a suitable family of interfaces supporting the right balance between abstraction and the export of essential operations. We believe that the treatment of anti-chains in this paper can serve as a guide to their use as an optimization in a wide range of circumstances.

We would like to thank Prakash Panangaden for starting us on the idea of exploring order-theoretic commonalities between AI representations. Elsa Gunter and Dave MacQueen provided useful help on the module interface design questions. Haym Hirsh was helpful to us in understanding the usefulness of abstractions for the VS algorithm. We also thank Ziqiang He, Sampath Kannan, Rona Machlin, and Bonnie Webber. Gunter's work was partially supported by AT&T Bell Laboratories and ONR. Ngair's work was supported by the Institute of Systems Science, Singapore. Subramanian's work was supported by NSF grant IRI-8902721.

References

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

[2] Carl A. Gunter, T-H. Ngair, P. Panangaden, and D. Subramanian. The common order-theoretic structure of version spaces and ATMS's. In *Ninth National Conference on Artificial Intelligence*, volume 1, pages 500–505, Anaheim CA, July 1991. AAAI Press.

[3] Haym Hirsh. *Incremental Version Space Merging: A General Framework for Concept Learning.* Kluwer Academic Publishers, 1990.

[4] C. Mellish. The description identification problem. *Artificial Intelligence*, 52:151–167, 1991.

[5] T. Mitchell. The need for biases in generalization. In J. Shavlik and T. Dietterich, editors, *Readings in Machine Learning.* Morgan Kaufmann, 1990.

[6] Tom Mitchell. *Version Space: An approach to Concept Leaning.* PhD thesis, Stanford University, 1978.

Signature for Anti-Chains

```
signature ANTICHAIN =
  sig
    type elt
    type ac
    exception NotFound
    val empty : ac
    val isEmpty : ac -> bool
    val singleton : elt -> ac (* Create a singleton ac *)
    val equal : (ac * ac) -> bool
    val numElts : ac ->  int (* Return the number of elt's in the ac *)
    val listElts : ac -> elt list
        (* Return a list of the elt's in the ac *)
    val app : (elt -> 'b) -> ac -> unit
        (* Apply a function to the elt's in the ac in decreasing order *)
    val revapp : (elt -> 'b) -> ac -> unit
        (* Apply a function to the elt's in the ac in increasing order *)
    val fold : (elt * 'b -> 'b) -> ac -> 'b -> 'b
        (* Fold in decreasing order *)
    val revfold : (elt * 'b -> 'b) -> ac -> 'b -> 'b
        (* Fold in increasing order *)
    val exists : (elt -> bool) -> ac -> elt option
        (* Return an elt in the ac satisfying the
         * predicate if any, return NONE if there is none *)
    val upper_add : ac * elt -> ac
    val lower_add : ac * elt -> ac (* Insert an elt *)
    val upper_find : ac * elt -> elt
    val lower_find : ac * elt -> elt
        (* Find an elt in an set, raise NotFound if not found *)
    val upper_peek : ac * elt -> elt option
    val lower_peek : ac * elt -> elt option
        (* Look for an elt in a set,
         * return NONE if the elt is not there. *)
    val upper_member : ac * elt -> bool
    val lower_member : ac * elt -> bool
        (* Return true iff elt is in the set *)
    val upper_subset : (ac * ac) -> bool
    val lower_subset : (ac * ac) -> bool
    val upper_difference : ac * ac -> ac
    val lower_difference : ac * ac -> ac
    val upper_union : ac * ac -> ac
    val lower_union : ac * ac -> ac
    val upper_homogeneous_intersection : ac * ac -> ac
    val lower_homogeneous_intersection : ac * ac -> ac
    val upper_heterogeneous_intersection : ac * ac -> ac
    val lower_heterogeneous_intersection : ac * ac -> ac
  end (* ANTICHAIN *)
end
```